

垃圾回收算法手册

自动内存管理的艺术

[英] 理查德·琼斯 (Richard Jones)
[美] 安东尼·霍思金 (Antony Hosking) 著 王雅光 薛迪 译
艾略特·莫斯 (Eliot Moss)

The Garbage Collection Handbook
The Art of Automatic Memory Management

THE GARBAGE COLLECTION HANDBOOK

The Art of Automatic Memory Management

Richard Jones
Antony Hosking
Eliot Moss

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

垃圾回收算法手册

自动内存管理的艺术

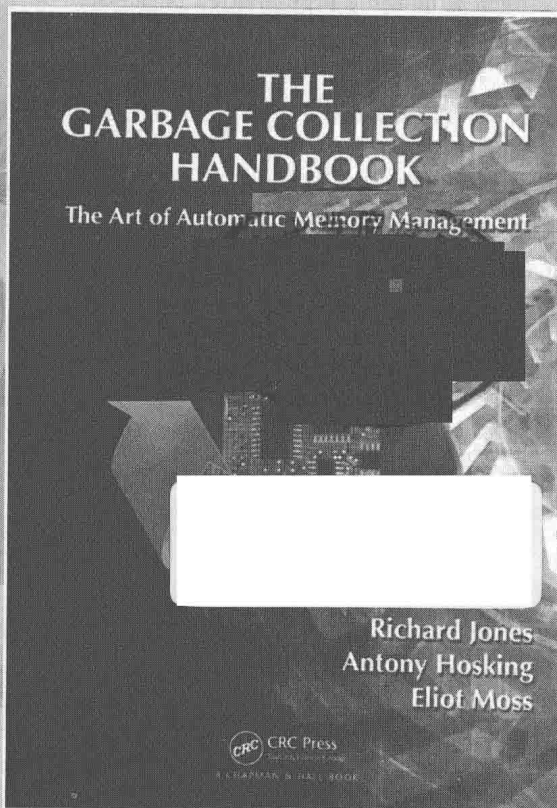
[英] 理查德·琼斯 (Richard Jones)

[美] 安东尼·霍思金 (Antony Hosking) 著 王雅光 薛迪 译

艾略特·莫斯 (Eliot Moss)

The Garbage Collection Handbook

The Art of Automatic Memory Management



机械工业出版社
China Machine Press

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅肇划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

垃圾回收技术给编程所带来的好处是不言而喻的，它能够从根本上解决软件开发过程中的内存管理问题，大大提升开发效率。但是目前，垃圾回收领域的书籍却非常少。

本书可以说是垃圾回收领域排名第二的经典著作之一，排名第一的也出自同一作者之手。1996年，Jones等的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 一书出版，并于2004年译成中文。回顾1996年，垃圾回收技术的大范围应用才刚刚起步——C++正称霸着软件开发领域，Java语言才推出一年之久，Anders Hejlsberg（C#之父，.NET的创立者）刚刚加入微软公司。近20年过去了，垃圾回收技术早已在各种编程语言中遍地开花，而且几乎成为每种新诞生语言的标配。与1996年的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 一书相比，本书不仅在内容上更加丰富，而且更加注重充分利用近20年来硬件发展所带来的机遇与挑战。

本书从最基础的垃圾回收算法出发，进一步介绍了到目前为止已经十分成熟的工业级垃圾回收技术实现（例如分代回收机制），这些内容基本上算是垃圾回收领域的“经典”内容。与此同时，面对多核技术的发展以及并程序的普及，本书使用接近一半的篇幅介绍了如何充分利用多处理器的能力来实现垃圾回收（例如并行回收、并发回收等），相关技术大都是近些年才研发出来的新成果，代表着垃圾回收领域最先进的发展方向。本书最后进一步介绍了垃圾回收技术在实时系统领域的最新研究成果。

对于开发人员而言，在享受垃圾回收机制所带来便利的同时，是否曾想过隐藏在它背后的秘密？在进行技术选型时，如何评估垃圾回收对性能可能造成的影响？面对编程语言所提供的种类繁多的垃圾回收相关参数，应当如何进行配置与调优？通过本书，开发人员能够更加深入地了解垃圾回收方面的相关问题、不同回收器的工作模式。对于研究生以及大学生而言，如果他们对编程语言的垃圾回收机制的技术实现感兴趣，本书将是不二之选。

最后，我要感谢那些在翻译过程中给予我帮助与支持的人。首先要感谢《Java虚拟机规范（Java SE 7版）》的译者薛迪将我引入技术书籍翻译这个领域，并在翻译过程中指出我的许多不足。同时还要感谢机械工业出版社的吴怡和张梦玲编辑对我的帮助，以及对我延迟交稿的包容。最后要特别感谢的是我的妻子，在这一年多的时间里，翻译工作让我没有太多的时间陪她，在此致以深深的歉意。

王雅光

2015年12月

1960年，McCarthy 和 Collins 发表了第一篇有关自动动态内存管理（即垃圾回收）的论文。弹指一挥间，50多年后的今天，本书也已截稿。垃圾回收机制诞生于 Lisp 程序语言，无巧不成书，Lisp 语言诞生于 1958 年，在其 40 周年之际，第一届国际内存管理研讨会（International Symposium on Memory Management）于 1998 年 10 月举办，而本书开始写作的时间也恰逢此次会议召开 10 周年。McCathy[1978] 回忆他在麻省理工学院工业联络研讨会上第一次现场演示 Lisp 语言时的情形，他们本想给观众留下良好的第一印象，但不幸的是，IBM 704[⊖]在演示的中途就耗尽了全部的 32KB 内存空间，电传打字机以每秒十个字符的速度输出

THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS:

以及其他一些更加冗长的错误信息，这一问题几乎占据了当时的整个演示时间，于是 McCarthy 的项目小组不得不省略刷新 Lisp 核心映像的相关内容，并在观众的笑声中无奈地结束演示。50 多年后的今天，垃圾回收早已不再是一个笑话，反而已经成为现代编程语言实现的关键组成部分之一。对于所有诞生于 1990 年之后且得到广泛应用的编程语言，Visual Basic（出现于 1991 年）是其中唯一一个没有采用自动内存管理的语言，但是其现代版本 VB.NET（出现于 2002 年）却依赖于具备垃圾回收能力的微软公共语言运行时（Microsoft common language runtime）。

垃圾回收给软件开发带来的收益不胜枚举。它可以消除开发过程中的几大类错误，例如尝试对悬挂指针（即指向已经回收或错误甚至被重新分配出去的内存）进行解引用，或者对已经释放的内存进行二次释放。尽管其不能保证完全消除内存泄漏问题，但也能大幅减少该问题的出现几率，还能够大幅简化并发数据结构的构建和使用 [Herlihy and Shavit, 2008]。综上所述，开发者能够基于垃圾回收所提供的抽象能力进行更好的软件工程实践。它简化了用户接口，使得代码更加容易理解和维护，进而更加可靠。由于用户接口不再需要关注内存管理，所以提升了代码的可复用性。

在过去的数年中，内存管理技术在软件和硬件方面都取得了长足进步。1996 年，典型的 Intel 奔腾处理器的时钟速度只有 120MHz，就连基于 Digital 的 Alpha 芯片的高端工作站主频也只有 266MHz。而在今天，主频达到 3GHz 以上的高端处理器以及多核芯片已经非常普遍，主存空间也几乎取得了 1000 倍的增长，普通台式计算机的内存大小已经从最初的几兆字节扩展到了 4GB。尽管如此，DRAM 内存的性能提升速度依然赶不上处理器的主频增长速度。我们曾在 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 中指出，“垃圾回收是能够解决所有内存管理问题的灵丹妙药”，并特别指出“垃圾回收机制尚无法应用于

⊖ IBM 704 为 Lisp 贡献了 car 和 cdr 这两个概念，并沿用至今。在 IBM 704 中，每个 36 位的字都包含一个地址部分（address part）以及一个减量部分（decrement part），它们均占用 15 位。Lisp 的链表或者 cons 单元将指针存储在这两部分中。链表头部或者 cons 单元的第二个元素可以使用 IBM 704 的 car（contents of the address part of register）指令获取；相应地，链表尾部或者 cons 单元的第一个元素可以使用 cdr（contents of the decrement part of register）指令获取。

硬实时系统（即系统必须在给定时限内对事件做出响应）”。但时至今日，硬实时垃圾回收器已经走出实验室并进入到商业应用领域。尽管现代垃圾回收器已经解决了大多数内存管理问题，但新硬件、新环境以及新应用的出现仍会在内存管理领域不断抛出新的问题与挑战。

致读者

本书试图将过去 50 多年间学者和开发者们在自动内存管理领域所积累的丰富经验加以总结。所涉文献数量庞大，在写作期间我们的在线资源库收集了多达 2500 条记录。在描述最重要的实现策略以及代表最先进水平的实现技术时，我们尽量在一个统一的、易于接受的框架内进行讨论与比较。我们特别注意使用统一的风格和术语来介绍相关的算法与概念，同时辅以伪代码和插图来描述具体细节。对于关乎性能的部分，我们特别注意对底层细节的描述，例如同步操作原语的选择、硬件组件（如高速缓存）对算法设计的影响。

在过去的 10 年间，硬件和软件设施的发展给垃圾回收领域带来了许多新的挑战。处理器和内存之间的性能差距总体在不断扩大。处理器时钟速度得到大幅增长，单个芯片上集成的处理器核心数量越来越多，使用多处理器的模块也越来越普遍。本书重点关注了这些变化对高性能垃圾回收器的设计与实现所造成的影响。由于高速缓存对性能的影响至关重要，所以垃圾回收算法必须考虑到局部性问题。越来越多的应用程序已经多线程化，且运行在多核处理器之上，因而我们应当避免内存管理器成为性能瓶颈。另外，垃圾回收器的设计应当充分利用硬件的并行能力。在 Jones[1996][⊖]中，我们完全没有考虑如何使用多线程进行并行回收，只用一章的篇幅来介绍增量回收与并发回收，这在当时的书中显得格外引人注目。

本书自始至终都密切关注现代硬件所带来的机遇与限制，对局部性问题的考量将贯穿全书。我们默认应用程序可能是多线程的。尽管本书涵盖了很多更加简单、更加传统的算法，但我们还是花了全书近一半的篇幅来介绍并行回收、增量回收、并发回收以及实时回收。

我们希望本书能够帮助到对编程语言实现感兴趣的研究生、研究人员和开发人员。对于选修了编程语言、编译器构建、软件工程或操作系统方面高级课程的本科生而言，本书也会有所帮助。此外，我们希望专业开发人员能够通过本书更加深入地了解垃圾回收面临的相关问题、不同回收器的工作模式，我们相信，与具体的专业知识相结合，开发人员在面对多种垃圾回收方法时，能够更好地进行回收器的选型与配置。由于几乎所有的现代编程语言都提供了垃圾回收机制，所以全面了解这一课题对所有开发者来说都是不可或缺的。

本书结构

本书第 1 章以探讨为什么需要自动内存管理作为开篇，简要介绍了对不同垃圾回收策略进行比较的方法。该章结尾介绍了贯穿全书的抽象记法与伪代码描述方式。

接下来的 4 章详细描述了 4 种经典的垃圾回收算法，分别是标记-清扫算法、标记-整理算法、复制式回收算法以及引用计数算法。本书对这些回收算法进行了深入的研究，并特别关注了其在现代硬件设施上的实现。如果读者需要一些更加基础的介绍，可以参阅我们先前的—本书 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* (Richard Jones and Rafael Lins, Wiley, 1996)。第 6 章深入比较了第 2 ~ 5 章所介绍的回收策略与算法，评估了它们各自的优缺点以及在不同情况下的适用性。

⊖ 指的是 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*。——编辑注

内存回收策略同样依赖于内存分配策略。第 7 章介绍了多种不同的内存分配技术，并进一步探究了自动垃圾回收与显式内存管理这两种场景下分配策略的不同之处。

前 7 章假定所有堆中的对象均采用相同的管理策略，但根据许多因素可知这并非一种良好的设计策略。第 8 章讨论了为何需要将堆划分为多个不同的空间，以及如何管理这些空间；第 9 章介绍了最成功的对象管理策略之一：分代垃圾回收；第 10 章介绍了大对象的管理策略以及其他分区策略。

在构建垃圾回收器的过程中，与运行时系统其他部分的对接是最复杂的内容之一[⊖]。因此第 11 章用了一整章的篇幅来介绍运行时接口，包括指针查找、能够安全发起垃圾回收的代码位置、读写屏障等。第 12 章讨论了特定语言相关内容，包括终结机制和弱引用。

在接下来的章节中，我们将注意力集中在并发环境下。第 13 章探讨了现代硬件系统给垃圾回收器的实现者所带来的新机遇与挑战，同时介绍了同步、前进、结束、一致等问题的相关算法。第 14 章介绍如何在挂起所有应用程序线程的前提下使用多个线程进行垃圾回收。接下来的 4 章介绍了多种不同种类的并发回收器，它们均放宽了“万物静止”这一要求，其回收过程只需要给用户程序引入十分短暂的停顿。最后，第 19 章探讨了最富挑战性的课题，即垃圾回收在硬实时系统中的应用。

每一章结尾都总结了一些需要考虑的问题，其目的在于引导读者去思考自己的系统有什么样的需求，以及如何满足这些需求，这些问题不仅关乎用户程序的行为，也关乎操作系统，甚至底层硬件的形为。但这些问题并不能替代对具体章节的阅读，它们并不是描述现有解决方案，而是提供进一步研究的焦点。

本书缺少了哪些内容？我们仅仅讨论了内嵌于运行时系统的自动内存管理技术，即使编程语言指定了垃圾回收相关的规范，我们也没有深入探讨其所支持的其他内存管理机制。最明显的例子是区域（region）的应用 [Tofte and Talpin, 1994]，其在 Java 实时规范中占据着显著的地位。我们仅花费了少量的篇幅来介绍区域推断以及栈上分配技术，并且几乎没有涉及其他通过编译期分析来替代，甚至辅助垃圾回收的技术。尽管引用计数策略在 C++ 等语言中得到了广泛应用，但我们依然认为它不是在用户程序中进行自动内存管理的最佳选择。最后，我们认为，下一代计算机将采用高度非一致内存架构，并配备异构垃圾回收器（heterogeneous collector）。这方面的技术与分布式垃圾回收（distributed garbage collection）的相关性较大，但在过去的数十年间，分布式垃圾回收领域鲜有新的研究成果发表，这不得不说不是一件憾事。本书没有介绍分布式垃圾回收的相关内容。

在线资源

本书相关的电子资料参见：<http://www.gchandbook.org>。

该网站包含了大量垃圾回收相关资源，包括本书完整的参考文献。本书末尾所列的参考文献超过了 400 条，但我们的在线数据库中有超过 2500 条垃圾回收相关文献。该数据库支持在线搜索，同时还支持 BibTeX、PostScript、PDF 格式的下载。除了相关文章、论文、书籍之外，该参考文献还包含了某些文献的摘要，对于大多数存在电子版的文献，我们还给出了相关 URL 以及 DOI 信息。

我们将持续更新本书参考文献，并将其作为一项社区服务。如果有更多文献（或者修正意见），欢迎联系 Richard (R.E.Jones@kent.ac.uk)，我们将不胜感激。

⊖ 我们在 Jones[1996] 中省略了相关内容。

致谢

感谢各位同事在本书编写时所给予的各项支持，没有大家的鼓励（与压力），本书的问世可能依然遥遥无期。特别需要感谢的是 Steve Blackburn、Hans Boehm、David Bacon、Cliff Click、David Detlefs、Daniel Frampton、Robin Garner、Barry Hayes、Laurence Hellyer、Maurice Herlihy、Martin Hirzel、Tomáš Kalibera、Doug Lea、Simon Marlow、Slan Mycroft、Cosmin Oancea、Erez Petrank、Fil Pizlo、Tony Printezis、John Reppy、David Siegart、Gil Tene 以及 Mario Wolczko，感谢诸位十分耐心地解答了我们的许多疑问，并对本书的草稿给予诸多有用的反馈意见。同时我们也在向 1958 年以来所有致力于自动内存管理研究的计算机科学家们致敬，没有他们的努力，本书也无从而来。

此外，我们还要向 Taylor and Francis 出版社的编辑 Randi Cohen 女士表示衷心的感谢，感谢她的支持和耐心。她总是能够及时地给予我们帮助，对我们的延误也展现出了最大程度的忍耐。同时还要感谢 Elizabeth Haylett 以及英国作家协会[⊖]的帮助，并极力向各位作者推荐他们。

Richard Jones、Antony Hosking、Eliot Moss

首先我要特别感谢 Robbie，本书从开始计划到最终完成，编写时间超过了预期，耗时两年，在此期间她忍受着我无法想象的巨大压力。本书的出版都归功于她！此外，如果没有另外两位合作者无尽的热情，本书是否能够问世恐怕还是很大一个问号。Tony 和 Eliot，很高兴也非常荣幸能与这两位勤奋博学的同事一起完成此书。

Richard Jones

2002 年的夏天，Richard 和我计划为他 1996 年的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 续写一部新书。在这 6 年当中，垃圾回收领域诞生了很多新的工作成果，因此有必要对前书的内容进行更新。当时我们不知道，这一本书的问世会再需要 9 年的时间，由此我不得不佩服 Richard 的耐心。当设想变成具体的计划时，我们荣幸邀请到 Eliot 加入本书的编写工作中，没有他的全力协助，我们现在可能还处在焦虑的工作之中。本书的前期计划与工作是在 Richard 与我在 2008 年休假期间展开的，且本书的编写工作受到了来自英国工程和物理科学研究委员会以及美国国家科学基金会的支持，我们在此表示感谢。在此，还要特别感谢 Mandi 的鼓励，感谢你同意我把大量的时间花费在这个项目之上，否则我是不可能完成这项工作的。

Antony Hosking

感谢另外两位合著者邀请我参与此项已经充分构思且已拟定出版的书籍编写项目。非常荣幸能与你们一同工作，也非常感谢你们能容忍我另类的写作风格。在此还要感谢英国皇家工程学院为我 2009 年 11 月的英国之行提供支持，这在很大程度上推进了此书的完成。除此之外，还要感谢其他基金会间接资助我们参加各种会议，并给予我们面对面交流的机会。最需要感谢的是我的妻子以及女儿能包容我出差或心思不在家庭上。她们的支持是最重要的，也是我最珍惜的！

Eliot Moss

⊖ <http://www.societyofauthors.org>。

目 录

The Garbage Collection Handbook: the Art of Automatic Memory Management

出版者的话

译者序

前言

作者简介

第 1 章 引言 1

1.1 显式内存释放 1

1.2 自动动态内存管理 3

1.3 垃圾回收算法之间的比较 5

1.3.1 安全性 5

1.3.2 吞吐量 5

1.3.3 完整性与及时性 5

1.3.4 停顿时间 6

1.3.5 空间开销 7

1.3.6 针对特定语言的优化 7

1.3.7 可扩展性与可移植性 8

1.4 性能上的劣势 8

1.5 实验方法 8

1.6 术语和符号 10

1.6.1 堆 10

1.6.2 赋值器与回收器 11

1.6.3 赋值器根 11

1.6.4 引用、域和地址 11

1.6.5 存活性、正确性以及可达性 12

1.6.6 伪代码 12

1.6.7 分配器 13

1.6.8 赋值器的读写操作 13

1.6.9 原子操作 13

1.6.10 集合、多集合、序列以及
元组 14

第 2 章 标记 - 清扫回收 15

2.1 标记 - 清扫算法 16

2.2 三色抽象 18

2.3 改进的标记 - 清扫算法 18

2.4 位图标记 19

2.5 懒惰清扫 21

2.6 标记过程中的高速缓存不命中问题 24

2.7 需要考虑的问题 25

2.7.1 赋值器开销 25

2.7.2 吞吐量 26

2.7.3 空间利用率 26

2.7.4 移动, 还是不移动 26

第 3 章 标记 - 整理回收 28

3.1 双指针整理算法 29

3.2 Lisp 2 算法 30

3.3 引线整理算法 32

3.4 单次遍历算法 34

3.5 需要考虑的问题 36

3.5.1 整理的必要性 36

3.5.2 整理的吞吐量开销 36

3.5.3 长寿数据 36

3.5.4 局部性 37

3.5.5 标记 - 整理算法的局限性 37

第 4 章 复制式回收 38

4.1 半区复制回收 38

4.1.1 工作列表的实现 39

4.1.2 示例 40

4.2 遍历顺序与局部性 42

4.3 需要考虑的问题 46

4.3.1 分配 46

4.3.2 空间与局部性 47

4.3.3 移动对象 48

第 5 章 引用计数 49

5.1 引用计数算法的优缺点 50

5.2 提升效率 51

5.3 延迟引用计数 52

5.4 合并引用计数 54

5.5 环状引用计数 57

5.6 受限域引用计数 61

5.7 需要考虑的问题 62

5.7.1 应用场景	62	8.2.3 为空间进行分区	88
5.7.2 高级的解决方案	62	8.2.4 根据类别进行分区	89
第 6 章 垃圾回收器的比较	64	8.2.5 为效益进行分区	89
6.1 吞吐量	64	8.2.6 为缩短停顿时间进行分区	90
6.2 停顿时间	65	8.2.7 为局部性进行分区	90
6.3 内存空间	65	8.2.8 根据线程进行分区	90
6.4 回收器的实现	66	8.2.9 根据可用性进行分区	91
6.5 自适应系统	66	8.2.10 根据易变性进行分区	91
6.6 统一垃圾回收理论	67	8.3 如何进行分区	92
6.6.1 垃圾回收的抽象	67	8.4 何时进行分区	93
6.6.2 追踪式垃圾回收	67	第 9 章 分代垃圾回收	95
6.6.3 引用计数垃圾回收	69	9.1 示例	95
第 7 章 内存分配	72	9.2 时间测量	96
7.1 顺序分配	72	9.3 分代假说	97
7.2 空闲链表分配	73	9.4 分代与堆布局	97
7.2.1 首次适应分配	73	9.5 多分代	98
7.2.2 循环首次适应分配	75	9.6 年龄记录	99
7.2.3 最佳适应分配	75	9.6.1 集体提升	99
7.2.4 空闲链表分配的加速	76	9.6.2 衰老半区	100
7.3 内存碎片化	77	9.6.3 存活对象空间与柔性提升	101
7.4 分区适应分配	78	9.7 对程序行为的适应	103
7.4.1 内存碎片	79	9.7.1 Appel 式垃圾回收	103
7.4.2 空间大小分级的填充	79	9.7.2 基于反馈的对象提升	104
7.5 分区适应分配与简单空闲链表 分配的结合	81	9.8 分代间指针	105
7.6 其他需要考虑的问题	81	9.8.1 记忆集	106
7.6.1 字节对齐	81	9.8.2 指针方向	106
7.6.2 空间大小限制	82	9.9 空间管理	107
7.6.3 边界标签	82	9.10 中年优先回收	108
7.6.4 堆可解析性	82	9.11 带式回收框架	110
7.6.5 局部性	84	9.12 启发式方法在分代垃圾回收中的 应用	112
7.6.6 拓展块保护	84	9.13 需要考虑的问题	113
7.6.7 跨越映射	85	9.14 抽象分代垃圾回收	115
7.7 并发系统中的内存分配	85	第 10 章 其他分区策略	117
7.8 需要考虑的问题	86	10.1 大对象空间	117
第 8 章 堆内存的划分	87	10.1.1 转轮回回收器	118
8.1 术语	87	10.1.2 在操作系统支持下的 对象移动	119
8.2 为何要进行分区	87	10.1.3 不包含指针的对象	119
8.2.1 根据移动性进行分区	87	10.2 基于对象拓扑结构的回收器	119
8.2.2 根据对象大小进行分区	88		

10.2.1 成熟对象空间的回收	120	11.8.6 卡表	172
10.2.2 基于对象相关性的回收	122	11.8.7 跨越映射	174
10.2.3 线程本地回收	123	11.8.8 汇总卡	176
10.2.4 栈上分配	126	11.8.9 硬件与虚拟内存技术	176
10.2.5 区域推断	127	11.8.10 写屏障相关技术小结	177
10.3 混合标记-清扫、复制式回收器	128	11.8.11 内存块链表	178
10.3.1 Garbage-First 回收	129	11.9 地址空间管理	179
10.3.2 Immix 回收以及其他回收	130	11.10 虚拟内存页保护策略的应用	180
10.3.3 受限内存空间中的复制式回收	133	11.10.1 二次映射	180
10.4 书签回收器	134	11.10.2 禁止访问页的应用	181
10.5 超引用计数回收器	135	11.11 堆大小的选择	183
10.6 需要考虑的问题	136	11.12 需要考虑的问题	185
第 11 章 运行时接口	138	第 12 章 特定语言相关内容	188
11.1 对象分配接口	138	12.1 终结	188
11.1.1 分配过程的加速	141	12.1.1 何时调用终结方法	189
11.1.2 清零	141	12.1.2 终结方法应由哪个线程调用	190
11.2 指针查找	142	12.1.3 是否允许终结方法彼此之间的并发	190
11.2.1 保守式指针查找	143	12.1.4 是否允许终结方法访问不可达对象	190
11.2.2 使用带标签值进行精确指针查找	144	12.1.5 何时回收已终结对象	191
11.2.3 对象中的精确指针查找	145	12.1.6 终结方法执行出错时应当如何处理	191
11.2.4 全局根中的精确指针查找	147	12.1.7 终结操作是否需要遵从某种顺序	191
11.2.5 栈与寄存器中的精确指针查找	147	12.1.8 终结过程中的竞争问题	192
11.2.6 代码中的精确指针查找	157	12.1.9 终结方法与锁	193
11.2.7 内部指针的处理	158	12.1.10 特定语言的终结机制	193
11.2.8 派生指针的处理	159	12.1.11 进一步的研究	195
11.3 对象表	159	12.2 弱引用	195
11.4 来自外部代码的引用	160	12.2.1 其他动因	196
11.5 栈屏障	162	12.2.2 对不同强度指针的支持	196
11.6 安全回收点以及赋值器的挂起	163	12.2.3 使用虚对象控制终结顺序	199
11.7 针对代码的回收	165	12.2.4 弱指针置空过程的竞争问题	199
11.8 读写屏障	166	12.2.5 弱指针置空时的通知	199
11.8.1 读写屏障的设计工程学	167	12.2.6 其他语言中的弱指针	200
11.8.2 写屏障的精度	167	12.3 需要考虑的问题	201
11.8.3 哈希表	169	第 13 章 并发算法预备知识	202
11.8.4 顺序存储缓冲区	170	13.1 硬件	202
11.8.5 溢出处理	172	13.1.1 处理器与线程	202

13.1.2 处理器与内存之间的互联	203	14.6.1 以处理器为中心的并行复制	254
13.1.3 内存	203	14.6.2 以内存为中心的并行复制技术	258
13.1.4 高速缓存	204	14.7 并行清扫	263
13.1.5 高速缓存一致性	204	14.8 并行整理	264
13.1.6 高速缓存一致性对性能的影响示例：自旋锁	205	14.9 需要考虑的问题	267
13.2 硬件内存一致性	207	14.9.1 术语	267
13.2.1 内存屏障与先于关系	208	14.9.2 并行回收是否值得	267
13.2.2 内存一致性模型	209	14.9.3 负载均衡策略	267
13.3 硬件原语	209	14.9.4 并行追踪	268
13.3.1 比较并交换	210	14.9.5 低级同步	269
13.3.2 加载链接/条件存储	211	14.9.6 并行清扫与并行整理	270
13.3.3 原子算术原语	212	14.9.7 结束检测	270
13.3.4 检测-检测并设置	213	第15章 并发垃圾回收	271
13.3.5 更加强大的原语	213	15.1 并发回收的正确性	272
13.3.6 原子操作原语的开销	214	15.1.1 三色抽象回顾	273
13.4 前进保障	215	15.1.2 对象丢失问题	274
13.5 并发算法的符号记法	217	15.1.3 强三色不变式与弱三色不变式	275
13.6 互斥	218	15.1.4 回收精度	276
13.7 工作共享与结束检测	219	15.1.5 赋值器颜色	276
13.8 并发数据结构	224	15.1.6 新分配对象的颜色	276
13.8.1 并发栈	226	15.1.7 基于增量更新的解决方案	277
13.8.2 基于单链表的并发队列	228	15.1.8 基于起始快照的解决方案	277
13.8.3 基于数组的并发队列	230	15.2 并发回收的相关屏障技术	277
13.8.4 支持工作窃取的并发双端队列	235	15.2.1 灰色赋值器屏障技术	278
13.9 事务内存	237	15.2.2 黑色赋值器屏障技术	279
13.9.1 何谓事务内存	237	15.2.3 屏障技术的完整性	280
13.9.2 使用事务内存助力垃圾回收器的实现	239	15.2.4 并发写屏障的实现机制	281
13.9.3 垃圾回收机制对事务内存的支持	240	15.2.5 单级卡表	282
13.10 需要考虑的问题	241	15.2.6 两级卡表	282
第14章 并行垃圾回收	242	15.2.7 减少回收工作量的相关策略	282
14.1 是否有足够多的工作可以并行	243	15.3 需要考虑的问题	283
14.2 负载均衡	243	第16章 并发标记-清扫算法	285
14.3 同步	245	16.1 初始化	285
14.4 并行回收的分类	245	16.2 结束	287
14.5 并行标记	246	16.3 分配	287
14.6 并行复制	254	16.4 标记过程与清扫过程的并发	288
		16.5 即时标记	289
		16.5.1 即时回收的写屏障	290

16.5.2	Doligez-Leroy-Gonthier 回收器	290	19.2	实时回收的调度	334
16.5.3	Doligez-Leroy-Gonthier 回收器在 Java 中的应用	292	19.3	基于工作的实时回收	335
16.5.4	滑动视图	292	19.3.1	并行、并发副本回收	335
16.6	抽象并发回收框架	293	19.3.2	非均匀工作负载的影响	341
16.6.1	回收波面	294	19.4	基于间隙的实时回收	342
16.6.2	增加追踪源头	295	19.4.1	回收工作的调度	346
16.6.3	赋值器屏障	295	19.4.2	执行开销	346
16.6.4	精度	295	19.4.3	开发者需要提供的信息	347
16.6.5	抽象并发回收器的实例化	296	19.5	基于时间的实时回收: Metronome 回收器	347
16.7	需要考虑的问题	296	19.5.1	赋值器使用率	348
第 17 章	并发复制、并发整理算法	298	19.5.2	对可预测性的支持	349
17.1	主体并发复制: Baker 算法	298	19.5.3	Metronome 回收器的分析	351
17.2	Brooks 间接屏障	301	19.5.4	鲁棒性	355
17.3	自删除读屏障	301	19.6	多种调度策略的结合: “税收与开支”	355
17.4	副本复制	302	19.6.1	“税收与开支”调度策略	356
17.5	多版本复制	303	19.6.2	“税收与开支”调度策略 的实现基础	357
17.6	Sapphire 回收器	306	19.7	内存碎片控制	359
17.6.1	回收的各个阶段	306	19.7.1	Metronome 回收器中 的增量整理	360
17.6.2	相邻阶段的合并	311	19.7.2	单处理器上的增量副本复制	361
17.6.3	Volatile 域	312	19.7.3	Stopless 回收器: 无锁垃圾回收	361
17.7	并发整理算法	312	19.7.4	Staccato 回收器: 在赋值器 无等待前进保障条件下的 尽力整理	363
17.7.1	Compressor 回收器	312	19.7.5	Chicken 回收器: 在赋值器无 等待前进保障条件下的尽力 整理 (x86 平台)	365
17.7.2	Pauseless 回收器	315	19.7.6	Clover 回收器: 赋值器乐观 无锁前进保障下的可靠整理	366
17.8	需要考虑的问题	321	19.7.7	Stopless 回收器、Chicken 回收 器、Clover 回收器之间的 比较	367
第 18 章	并发引用计数算法	322	19.7.8	离散分配	368
18.1	简单引用计数算法回顾	322	19.8	需要考虑的问题	370
18.2	缓冲引用计数	324	术语表		372
18.3	并发环境下的环状引用 计数处理	326	参考文献		383
18.4	堆快照的获取	326	索引		413
18.5	滑动视图引用计数	328			
18.5.1	面向年龄的回收	328			
18.5.2	算法实现	328			
18.5.3	基于滑动视图的环状 垃圾回收	331			
18.5.4	内存一致性	331			
18.6	需要考虑的问题	332			
第 19 章	实时垃圾回收	333			
19.1	实时系统	333			

引言

托管语言 (managed language) 以及托管运行时系统 (managed run-time system) 不仅能够提升程序的安全性, 而且可以通过对操作系统和硬件架构的抽象来提升代码的灵活性, 因而受到越来越多开发者的青睐。托管代码 (managed code) 的优点已经得到广泛认可 [Butters, 2007]。虚拟机 (virtual machine) 本身提供的多种服务可以减少开发者的工作量, 如果编程语言是类型安全的 (type-safe), 并且运行时系统可以在程序加载时进行代码检查, 在运行时对资源访问冲突、数组以及其他容器进行边界检查, 同时提供自动内存管理能力, 那么代码的安全性将更有保障。尽管“一次编译, 到处运行” (write once, run anywhere) 的说法有些言过其实, 但虚拟机确实使跨平台程序开发变得更加简单、开发成本更低。开发者也可将主要精力专注在应用程序的逻辑上。

几乎所有的现代编程语言都使用动态内存分配 (allocation), 即允许进程在运行时分配或者释放无法在编译期确定大小的对象, 且允许对象的存活时间超出创建这些对象的子程序时间[⊖]。动态分配的对象存在于堆 (heap) 中而非栈 (stack) 或者静态区 (statically) 中。所谓栈, 即程序的活动记录 (activation record) 或者栈帧 (stack frame); 静态区则是指在编译期或者链接期就可以确定范围的存储区域。堆分配是十分重要的功能, 它允许开发者:

- 在运行时动态地确定新创建对象的大小 (从而避免程序在运行时遭遇硬编码数组长度不足产生的失败)。
- 定义和使用具有递归特征的数据结构, 例如链表 (list)、树 (tree) 和映射 (map)。
- 向父过程返回新创建的对象, 例如工厂方法。
- 将一个函数作为另一个函数的返回值, 例如函数式语言中的闭包 (closure) 或者悬挂 (suspension)[⊖]。

堆中分配的对象需要通过引用 (reference) 进行访问。一般情况下, 引用即指向对象的指针 (pointer) (也就是对象在内存中的地址)。引用也可以通过间接方式实现, 例如它可以是一个间接指向对象的句柄 (handle)。使用句柄的好处在于, 当迁移某一对象时, 可以仅修改对象的句柄而避免通过程序改变这个对象或句柄的所有引用。

1

1.1 显式内存释放

任何一个运行在有限内存环境下的程序都需要不时地回收运行过程中不再需要的对象。堆中对象使用的内存可以使用显式释放 (explicit deallocation) 策略 (例如 C 语言的 free 函数或者 C++ 的 delete 操作符)、基于引用计数的运行时系统 [Collins, 1960]、追踪式垃圾回收器 [McCarthy, 1960] 进行回收。显式内存释放会在两个方面增加程序出错的风险。

一方面, 开发者可能过早地回收依然在引用的对象, 这种情况将引发悬挂指针 (dangling pointer) 问题 (如图 1.1 所示)。正常情况下, 开发者将不会访问已经释放的内存,

⊖ 本书中, 方法 (method)、函数 (function)、过程 (procedure)、子程序 (subroutine) 这几个概念等价。

⊖ 即待计算的表达式。——译者注

所以运行时系统可以将其清空（填充 0），也可以将其重新分配出去，甚至可以将其归还给操作系统。因此，如果程序对悬挂指针进行访问，那么执行结果将是不可预知的。此时开发者最期望的结果是程序立即崩溃，但更常见的情况却是程序会在崩溃前继续运行一段时间（导致调试困难），或者一直运行下去但输出错误的结果（甚至这些错误很难被检测到）。检测悬挂指针的一种策略是使用肥指针（fat pointer），它将指针目标对象的版本号作为指针本身的一部分。当对肥指针进行解引用时，运行系统会首先判断肥指针中记录的版本号与其目标对象的版本号是否一致。这一方法存在额外开销，而且并非完全可靠，因此其应用范围几乎局限于调试工具上^①。

另一方面，开发者很可能在程序将对象使用完毕之后未将对象释放，从而导致内存泄漏（memory leak）现象的出现。在小程序中，内存泄漏可能不会造成太大影响，但对于较大的程序，内存泄漏却有可能导致显著的性能下降（即内存管理器难以满足新的内存分配需求），甚至是崩溃（即程序的内存被耗光）。一次错误的内存释放通常不仅会导致悬挂指针的出现，而且会引发内存泄漏（如图 1.1 所示）。

上述两种错误在对象共享的情况下尤为普遍，即当两个或更多子过程持有同一个对象的引用时。在并发编程情况下，当两个或多个线程引用同一个对象时问题将更加严重。

随着多核处理器的普及，开发者需要投入相当多的精力来构建线程安全的数据结构库。实现线程安全的数据结构库的算法必须面临一系列问题，例如死锁（deadlock）、活锁（livelock）、ABA 问题（ABA problem）^②。自动内存管理可以显著降低这些问题的解决难度（例如可以消除某些情况下的 ABA 问题），否则编程方案将十分复杂 [Herlihy and Shavit, 2008]。

更为根本的问题在于，显式释放需要花费开发者更多的精力。如何正确进行内存管理往往是编程中的固有难题^③，正如 Wilson[1994]所指出的“存活性是一个全局（global）特征”，但是调用 free 函数将对象释放却是局部行为，所以如何将对象正确地释放是一个十分复杂的问题。

在不支持自动化动态内存管理的语言中，众多研究者已经付出了相当大的努力来解决这一难题，其方法主要是管理对象的所有权（ownership）[Belotsky, 2003；Cline, Lomow, 1995]。Belotsky[2003]等人针对 C++ 提出了几个可行策略：第一，开发者在任何情况下都应当避免堆分配，例如可以将对象分配在栈上，当创建对象的函数返回之后，栈的弹出（pop）操作会自动将对象释放。第二，在传递参数与返回值时，应尽量传值而非引用。尽管这些方法避免了分配、释放错误，但其不仅会造成内存方面的压力，而且失去了对象共享的能力。另外，开发者也可以在一些特殊场景下使用自定义内存分配器，例如对象池（pool of object），在程序的一个阶段完成之后，池中的对象将作为一个整体全部释放。

C++ 语言尝试通过特殊的指针类和模板来改善内存管理。此类方法通过对指针操作进行重载（overload）来提升内存回收的安全性，但是这些智能指针（smart pointer）通常存在

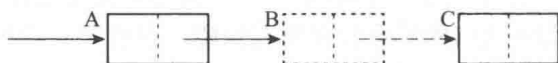


图 1.1 过早地删除对象可能引发错误。此处对象 B 被错误地释放，从而导致对象 A 中产生悬挂指针，且对象 C 所占用的空间发生泄漏，即对象 C 不可达，但无法将其释放

① 开源工具 valgrind (<http://valgrind.org>) 框架中使用的内存泄漏检测工具 memcheck 虽然速度较慢，但是更加可靠。另外还有很多商业化的内存调试工具。

② ABA 问题：一个变量的初始值是 A，被改写为 B，然后再次被改写为 A（参见第 13 章）。

③ “如果你手里拿着 C++ 这把锤子，那么所有东西看起来都像钉子”，Steven M. Haflich, Common Lisp ANSI 标准 NCITS/J13 技术委员会主席。

一些限制。`auto_ptr`与标准库不兼容,并且即将在下一个C++标准版本中废弃[Boehm, Spertus, 2009]^①,取而代之的是更加先进的`unique_ptr`,它将提供严格的所有权控制语义,即在该指针被摧毁时,其目标对象将自动得到释放。新标准也将提供一个基于引用计数的`shared_ptr`^②,但其仍存在一定限制,即引用计数指针无法处理自引用(环状引用)数据结构。大多数智能指针都是以库的形式提供,因此当需要关注性能时,其适用范围可能会受到限制。智能指针可能只适用于管理数据块非常大、引用关系变更较少的场景,因为只有在这种情况下,智能指针的开销才有可能远小于追踪式垃圾回收。另外,在没有编译器和运行时系统支持的情况下,基于引用计数的智能指针算不上是一种高效的、通用的小对象管理策略,特别是在指针操作非线程安全的情况下。

若安全地进行手动内存管理有过多的方法也会引发另一个问题:如果开发者始终需要考虑对象的所有权管理问题,那么他究竟应当使用哪种方法?当使用第三方代码库时,这一问题将更加严重。例如,第三方代码库使用哪种方法进行内存管理?所有的第三方库是否都使用同一种方法?

1.2 自动动态内存管理

自动动态内存管理可以解决大多数悬挂指针和内存泄漏问题。垃圾回收(garbage collection, GC)可以将未被任何可达对象引用的对象回收,从而避免悬挂指针的出现。原则上讲,回收器最终都会将所有不可达对象回收,但是有两个注意事项:第一,追踪式回收(tracing collection)引入“垃圾”这一具有明确判定标准的概念,但它不一定包含所有不再使用的对象;第二,后面章节将要描述,在实际情况下,出于效率原因,某些对象可能不会被回收。只有回收器可以释放对象,所以不会出现二次释放(double-freeing)问题。回收器掌握堆中对象的全局信息以及所有可能访问堆中对象的线程信息,因而其可以决定任意对象是否需要回收。显式释放的主要问题在于其无法在局部上下文中掌握全局信息,而自动动态内存管理则简单地解决了这一问题。

3

总之,内存管理是一个软件工程学问题。设计良好的程序应当是由一系列“高内聚,低耦合”的组件(从广义上来讲)构建而成。“高内聚”可以简化程序的维护,即在理想情况下,开发者如需理解一个模块,只需要关注该模块本身的代码或者其他相关模块的少量代码;“低耦合”意味着一个模块不依赖其他模块的实现。在这种情况下,如果要考虑正确的内存管理,就意味着一个模块要了解其他模块的内存管理规则。相比之下,显式内存管理显然不能满足软件工程学中“低耦合”的原则,它需要引入一些额外的接口,例如需要显式传递额外参数来协商对象的所有权,或者隐式要求开发者遵从一些特定惯例,这些要求都限制了模块的可复用性。

垃圾回收可以带来的好处绝不仅仅是简化编码,它同时可以解除模块之间内存管理层次的耦合,且不需要额外的内存管理接口,从而提升了模块的可复用性。正是由于这些原因,垃圾回收机制几乎成为现代编程语言的标配(如表1.1所示),甚至下一代C++标准都有可能引入垃圾回收机制^③[Boehm and Spertus, 2009]。大量证据表明,包含自动内存管理机制

① 当前,下一代符合ISO C++标准的最终提案是C++0x(该标准最终在2011年通过,被称为C++11)。——译者注

② 见<http://boost.org>。

③ 作者这里提到的“下一代”其实就是C++ 11标准,但该标准最终并未引入垃圾回收机制。——译者注

的托管代码可以降低开发成本 [Butters, 2007], 但不幸的是, 这些证据大多是未经证实的, 或者只是不同语言、不同系统之间的比较 (因此比较结果可能受到内存管理策略之外因素的影响), 更加详细的比较研究则少有表明这一观点。还有人建议将垃圾回收机制作为复杂系统中软件设计的主要关注点 [Nagle, 1995]。Rovner[1985] 估计, 在施乐公司 Mesa 系统的开发过程中, 40% 的开发时间花费在了内存管理相关问题的调试上。或许, 层出不穷的内存错误检测工具能够在经济方面间接地给予自动内存管理最有力的支持。

表 1.1 现代编程语言与垃圾回收 (这些语言都是基于垃圾回收的)

ActionScript (2000 年)	AppleScript (1993 年)	Beta (1983 年)
Managed C++ (2002 年)	Clean (1984 年)	Dylan (1992 年)
Eiffel (1986 年)	Erlang (1990 年)	Fortress (2006 年)
Groovy (2004 年)	Icon (1977 年)	Liana (1991 年)
Lisp (1958 年)	LotusScript (1995 年)	MATLAB (20 世纪 70 年代)
ML (1990 年)	Objective-C (2007 ~ 至今)	Pike (1996 年)
POP-2 (1970 年)	Python (1991 年)	Sather (1990 年)
Self (1986 年)	SISAL (1983 年)	Squeak (1996 年)
VB.NET (2001 年)	VHDL (1987 年)	Algol-68 (1965 年)
AspectJ (2001 年)	C# (1999 年)	Cecil (1992 年)
CLU (1974 年)	Dynace (1993 年)	Elasti-C (1997 年)
Euphoria (1993 年)	Green (1998 年)	Haskell (1990 年)
Java (1994 年)	Limbo (1996 年)	Lua (1994 年)
Mercury (1993 年)	Modula-3 (1988 年)	Obliq (1993 年)
PHP (1995 年)	PostScript (1982 年)	Rexx (1979 年)
Scala (2003 年)	SETL (1969 年)	Smalltalk (1972 年)
Tel (1990 年)	VBScript (1996 年)	X10 (2004 年)
APL (1964 年)	Awk (1977 年)	Cyclone (2006 年)
Cedar (1983 年)	D (2007 年)	E (1997 年)
Emerald (1988 年)	F# (2005 年)	Go (2010 年)
Hope (1978 年)	JavaScript (1994 年)	Lingo (1991 年)
Mathematica (1987 年)	Miranda (1985 年)	Oberon (1985 年)
Perl (1986 年)	Pliant (1999 年)	Prolog (1972 年)
Ruby (1993 年)	Scheme (1975 年)	Simula (1964 年)
SNOBOL (1962 年)	Theta (1994 年)	Visual Basic (1991 年)
YAFL (1993 年)		

诚然, 并不是说垃圾回收是根除所有与内存相关的编程错误且适用于所有场景的银弹 (silver bullet)。内存泄漏是最普遍的一种内存错误, 尽管垃圾回收可以减少内存泄漏现象的出现, 但也不能保证完全根除。如果某个对象在程序未来的运行过程中不可达 (例如, 从已知根集合开始通过任何指针链都不可达), 则回收器会将其回收, 这是删除对象的唯一方法, 因此悬挂指针不会出现; 如果删除某个对象导致其子对象也不可达, 则它也将得到回收, 所以图 1.1 所示的任何一种情况都不会出现。但是, 垃圾回收仍不能确保根除内存泄漏, 对于那种一直可达但无限增长的数据结构 (例如不停地将数据添加到缓冲区中, 但却不从中移除任何对象), 或者一直可达但永远不会再用到的对象, 垃圾回收也无能为力。

自动动态内存管理的设计目标仅限于其字面所表达的内容。一些评论者抱怨垃圾回收器不能提供通用的资源管理功能, 例如关闭不再使用的文件或者窗口, 但这一批评有失公允:

垃圾回收不是一剂万能的灵丹妙药，它所针对和解决的是一个特定的问题，即内存资源的管理问题。在具有垃圾回收功能的语言中，通用资源管理仍然是一个很大的问题。在显式内存管理的系统中，内存的释放与其他资源的释放具有直接且天然的联系，而自动内存管理系统却引入了新的问题：如何在缺少这种天然联系的情况下对其他资源进行管理。有趣的是，许多资源释放场景都会用到一些与垃圾回收类似的机制，并据此来检测某个资源在程序的后续运行过程中是否会继续使用（可达）。

1.3 垃圾回收算法之间的比较

本书介绍了多种不同类型的垃圾回收器，它们针对不同的工作负载、硬件环境以及性能要求而设计，但没有哪种回收器能够保证在所有情况下都有最优表现。例如，Fitzgerald 和 Tarditi[2000] 通过对 20 个基准测试程序以及 6 种回收器进行研究发现，当配备更加合理的回收器时，至少有一个基准程序的性能可以提升 15%；Singer 等 [2007b] 使用机器学习技术来预测适用于特定程序的最优垃圾回收器；还有一些研究者认为不同的回收器在不同特征的负载下会有不同表现，进而探究 Java 虚拟机（Java Virtual machine）在运行时切换回收器的方法，以期得到性能上的提升 [Printezis, 2001；Soman 等，2004]。本节我们将对在回收器之间进行比较的标准进行分析，但不论是在理论上还是在实践中，进行这样的比较通常都较为困难：具体的实现细节、算法的局部性（locality）、算法复杂度公式中常数的实际意义都会导致比较的结果对实际应用并不具有较高的指导价值；算法的性能不仅取决于对象在堆中的大小以及拓扑结构（topology），还与程序的访问模式（access pattern）有关；用于比较的各项指标之间并非相互独立，实际虚拟机中的各种优化选项也都具有一定内在联系，因此，针对特定目的对某一参数进行调整可能会对其他指标产生负面作用。

5

1.3.1 安全性

垃圾回收器首先要考虑的因素是安全性（safety），即在任何时候都不能回收存活对象。但安全性是需要付出一定代价的，特别是在并发回收器中（参见第 15 章）。对于不依赖编译器或者运行时系统的保守式回收（conservation collection），其安全性在理论上可能会被编译器的某些掩盖指针的优化所影响 [Jones, 1996，见第 9 章]。

1.3.2 吞吐量

对程序的最终用户而言，程序当然是运行得越快越好，但这是由几方面因素决定的。其中的一方面便是花费在垃圾回收上的时间应当越少越好，文献中通常用标记 / 构造率（mark/cons ratio）来衡量这一指标。这一概念是在早期的 Lisp 语言中最先提出的，它表示回收器（对存活对象进行标记）与赋值器（mutator）（创建或者构造新的链表单元）活跃度的比值。然而在大多数设计良好的架构中，赋值器会比回收器占用更多的 CPU 时间，因此在适当牺牲回收器效率的基础上提升赋值器的吞吐量，并进一步提升整个程序（赋值器 + 回收器）的执行速度，一般来说是值得的。例如，使用标记—清扫回收的系统偶尔会执行存活对象整理以减少内存碎片，虽然这一操作开销较大，但它可以提升赋值器的分配性能。

1.3.3 完整性与及时性

理想情况下，垃圾回收过程应当是完整的，即堆中的所有垃圾最终都应当得到回收，但

这通常是不现实的，甚至是不可取的，例如纯粹的引用计数回收器便无法回收环状引用垃圾（自引用结构）。从性能方面考虑，在一次回收过程（collection cycle）中只处理堆中部分对象或许更加合理，例如分代回收器会依照堆中对象的年龄将其划分为两代或者更多代（我们将在第9章描述分代垃圾回收），并把回收的主要精力集中在年轻代，这样不仅可以提高回收效率，而且可以减少单次回收的平均停顿时间（pause time）。

在并发垃圾回收器中，赋值器与回收器同时工作，其目的在于避免或者尽量减少用户程序的停顿。此类回收器会遇到浮动垃圾（floating garbage）问题，即如果某个对象在回收过程启动之后才变成垃圾，那么该对象只能在下一个回收周期内得到回收。因此在并发回收器中，衡量完整性更好的方法是统计所有垃圾的最终回收情况，而不是单个回收周期的回收情况。不同的回收算法在回收及时性（promptness）方面存在较大差异，进而需要在时间和空间上进行权衡。

6

1.3.4 停顿时间

许多回收器在进行垃圾回收时需要中断赋值器线程，因此会导致在程序执行过程中出现停顿。回收器应当尽量减少对程序主要执行过程的影响，因此要求停顿时间越短越好，这一点对于交互式程序或者事务处理服务器（超时将引发事务的重试，进而导致事务的积压）尤为重要。但正如我们在后面章节中将要看到的，限制停顿时间会带来一些副作用。例如，分代式回收器通过频繁且快速地回收较小的、较为年轻的对象来缩短停顿时间，而对较大的、较为年老对象的回收则只是偶尔进行。显然，在对分代回收器进行调优时，需要平衡不同分代的大小，进而才能平衡不同分代之间的停顿时间与回收频率。但由于分代回收器必须记录一些分代间指针的来源，因此赋值器的指针写操作会存在少量的额外开销。

并行回收器（parallel collector）虽然也需要停顿整个程序，但它可以通过多线程回收的策略缩短停顿时间。为进一步减少停顿时间，并发回收器与增量回收器（incremental collector）偶尔会将部分回收工作与赋值器动作交替进行或者同时进行，但这一过程需要确保赋值器与回收器之间的同步，因而增大了赋值器的额外开销。在第15章我们将看到，赋值器与回收器之间的同步存在多种实现方式。回收机制的选择会影响程序在空间和时间两个方面的开销，也会影响垃圾回收周期的结束。赋值器在时间方面的额外开销取决于需要记录的赋值器操作类型（读或者写）及其如何记录。回收器在空间方面的开销以及回收周期的结束取决于系统可以容忍的浮动垃圾数量。多线程赋值器与回收器会增大设计复杂度。不论如何，缩短停顿时间的措施通常会增大整体处理时间（即降低整体处理速度）。

仅对最大或者平均停顿时间进行度量是不够的，必须要同时考虑赋值器的性能，因此停顿时间的分布也值得关注。有多种方法可以描述停顿时间的分布，最简单的方法是统计停顿时间的变化，例如标准差或者图形表示等。更有效的方法包括最小赋值器使用率（minimum mutator utilization, MMU）和界限赋值器使用率（bounded mutator utilization, BMU）。MMU [Cheng and Blelloch, 2001] 和 BMU [Sachindran 等, 2004] 都简明地展示了任意给定时间窗内赋值器占用的（最小）时间比例。图 1.2 中， x 轴表示程序从开始到结束的整体执行时间， y 轴表示赋值器占用的 CPU 时间比例（使用率）^①。MMU 和 BMU 曲线不仅反映了垃圾回收

① 对于 MMU 曲线，假设当 $x=100$ 时， $y=50\%$ ，其含义是：在任意一个 100ms 的时间窗内，赋值器使用率的最小值是 50%；对于 BMU 曲线，假设当 $x=200$ 时， $y=60\%$ ，其含义是：在任意一个不小于 200ms 的时间范围内，赋值器使用率的最小值是 60%。——译者注

过程占整个执行时间的比例 (y 轴截距, 即曲线最右侧的点, 代表了赋值器占用整体处理时间的比例, 用 100% 减去该值即可得到回收过程的时间占比), 也反应了垃圾回收的最长停顿时间 (x 轴截距, 即赋值器 CPU 使用率为 0% 的最大时间窗口)。一般来说, 曲线在 y 轴上越高, 表示赋值器的 CPU 占用率越高, 在 x 轴上越靠左, 表示垃圾回收的最大停顿时间越小。MMU 曲线反映了程序在任意时间窗 (x) 内赋值器的最小使用率 (y), 但是较大时间窗的 MMU 可能会比较小时间窗的更低, 因此 MMU 曲线会出现下降。相比之下, BMU 曲线则反映出某一时间窗或者更大时间窗内的 MMU 值, 因此它是单调递增的。BMU 曲线或许比 MMU 曲线更为直观。

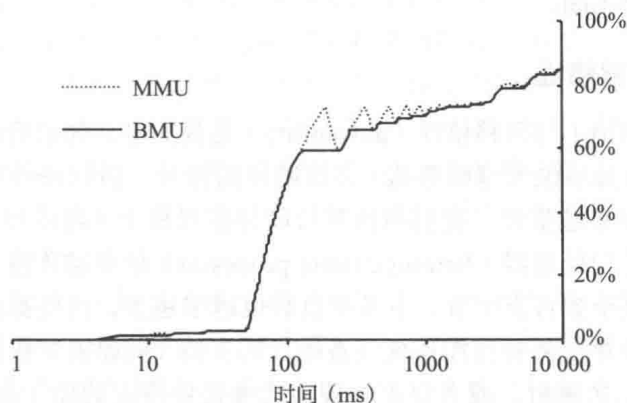


图 1.2 最小赋值器使用率 (MMU) 和界限赋值器使用率 (BMU) 曲线精确反映了任意时间窗内赋值器占用的 (最小) 时间比例。MMU 反映了任意给定时间窗 (x) 内的最小赋值器使用率 (y); BMU 反映了任意给定时间窗或者更大的时间范围内的最小赋值器使用率。在两种情况下, x 轴截距都反映最大停顿时间, y 轴截距反映赋值器占用整体时间的比例

1.3.5 空间开销

内存管理的目的是安全且高效地使用内存空间。不论是显式内存管理还是自动内存管理, 不同的管理策略均会产生不同程度的空间开销 (space overhead)。某些垃圾回收器需要在每个对象内部占用一定的空间 (例如保存引用计数), 还有一些回收器会复用对象现有布局上已经存在的域 (例如将标记位放在对象头部的某个字中, 或者将转发指针 (forwarding pointer) 记录在用户数据上)。回收器也可能会引入堆级别的空间开销, 例如复制式回收器需要将堆分为两个半区, 任何时候赋值器只能使用一个半区, 另一个半区会被回收器保留, 并在回收过程中将存活对象复制到其中。回收器也可能需要一些辅助的数据结构, 例如追踪式回收器需要通过标记栈来引导堆中指针图表的遍历, 回收器在标记对象时也可以使用额外的位图 (bitmap) 而非对象中的域; 对于并发回收器或者其他需要将堆划分为数个独立区域的回收器, 其需要额外的记忆集 (remembered set) 来保存赋值器所修改的指针值或者跨区域指针的位置。

1.3.6 针对特定语言的优化

垃圾回收算法可以根据它们所服务的不同语言范式来归类。在函数式语言中, 内存管理有着很大的优化空间。某些语言 (例如 ML) 将可变数据与不可变数据进行区分, 纯函数式语言 (例如 Haskell) 则更为极端, 它不允许用户改变任何数据, 即程序是透明引用

(referentially transparent) 的[⊖]。然而，在函数式语言内部，数据结构的更新一般不超过一次，即从待计算值 (thunk) 到一个弱头部范式 (weak head normal form, WHNF)[⊖]，分代垃圾回收器可以据此尽快提升已经完成计算的数据结构 (见第9章)。研究者们还提出了基于引用计数来处理环状数据结构的完整机制。声明式语言 (declarative language) 或许还可以使用其他策略来提升堆空间管理的效率，即如果某一对象创建于一个“选择点” (choice point) 之后，那么当程序再次回到该选择点时，该对象将不可达，如果对象在堆中的布局是按照其分配时间排布的，那么某个选择点之后分配的内存可以在一个固定的时间内全部回收。不同种类的语言可能对回收器具有不同的要求，最显著的差异是语言中指针功能的不同，以及回收器调用对象终结的需求不同。

1.3.7 可扩展性与可移植性

可扩展性 (scalability) 与可移植性 (portability) 是我们定义的最后两个指标。随着 PC，甚至笔记本计算机中 (且不说大型服务器) 多核硬件的普及，借助硬件的并行优势来提升垃圾回收的性能将变得越来越重要。我们期待并行硬件在规模上 (内核与套接字数量上) 能有进一步发展，也希望异构处理器 (heterogeneous processor) 越来越普遍。在服务器方面，堆的大小可以达到数十甚至数百吉字节，事务型负载也越来越多，这些都给垃圾回收带来更多的要求。很多垃圾回收算法需要操作系统或者硬件的支持 (例如需要依赖页保护机制，需要对虚拟内存空间进行二次映射，或者要求处理器能够提供特定的原子操作)，但这些技术并不需要很强的可移植性。

1.4 性能上的劣势

与显式内存管理相比，自动内存管理是否存在性能上的劣势？我们将通过对这一问题的分析，来对两者的优劣进行总结。一般来说，自动内存管理的运行开销很大程度上取决于程序的行为，甚至硬件条件，因而很难对其进行简单评估。一个长期以来的观点是，垃圾回收通常会在总内存吞吐量以及垃圾回收停顿时间方面引入一些不可接受的开销，从而导致应用程序的执行速度慢于显式内存管理策略。自动内存管理确实会牺牲程序的部分性能，但是远不如想象中那样严重。诸如 malloc 和 free 等显式内存操作也会带来一些显著开销。Herts, Feng, Herger[2005] 测量了多种 Java 基准测试程序和回收算法花费在垃圾回收上的真正开销。他们构建了一个 Java 虚拟机并用其精确地观察到对象何时不可达，同时使用可达追踪的方法驱动模拟器来测量回收周期与高速缓存不命中 (cache miss) 的情况。他们将许多不同种类的垃圾回收器配置与各种不同的 malloc/free 实现进行比较，比较的方法是：如果追踪发现某一对象变成垃圾，则调用 free 将其释放。Herts 等人发现，尽管用这两种方式的测量结果差异较大，但是如果堆足够大 (达到所需最小空间的 5 倍)，那么垃圾回收器的执行时间性能将可以与显式分配相匹敌，但对于一般大小的堆，垃圾回收的开销会平均增大 17%。

1.5 实验方法

在过去的数十年中，最最令人欣慰的变化之一是内存管理方面的各种文献报告推进了垃

⊖ 所谓透明引用，即以相同的参数调用同一个函数两次，所得到的结果总是相同的，也可理解为函数没有副作用。——译者注

⊖ thunk 和 WHNF 均为函数式语言中的与懒惰计算相关的概念。——译者注

圾回收实验方法的发展,但与自然科学或者社会科学相比,计算机科学的报告标准在质量方面仍有待提高。Mytkowicz 等 [2008] 发现,测量的偏差是“显著而又普遍”的。

Georges 等 [2007] 通过对大量垃圾回收方面的文献进行研究发现,垃圾回收实验方法(甚至是已经发表的)在许多场景下都不够严谨,许多已经发表的文献在性能方面的提升都非常小,且缺乏统计学分析,使得其结果缺乏可信度。实验误差的引入可能是系统性的,也可能是随机的。系统误差大多是由于实验不足导致的,通常可以通过更加仔细的实验设计来避免;随机误差通常是由于测量环境中系统的不确定性导致的,就其性质而言,这些错误通常不可预知,且经常超出实验者的控制范围,因而应当从统计学角度对其进行处理。

基于人造的小规模“玩具”基准测试程序进行实验的方法,由于其不能充分模拟真实环境,长期以来遭到批评 [Aorn, 1989]。这些实验基准测试程序不仅无法反映真实程序的内存分配交互情况,而且由于其工作集较小,以至于真实程序的局部性作用得不到完整表现,因而容易导致系统误差的出现, Wilson 等 [1995a] 对这种方法提出了很好的批判。除了压力测试场景,实验者们已经集体抛弃了这种人造的“玩具”基准测试程序,转而使用更大规模的基准测试程序套件,后者能够在更大范围上重现真实应用程序的行为(例如 Java 的 DaCapo 套件 [Blackburn 等, 2006b])。

基于具有大量真实程序案例的基准测试程序套件进行实验仍有可能引入系统性偏差,因为托管运行时在某些情况下也会引入一些系统误差。实验者必须很小心地区分他们所要测试的内容,即他们所感兴趣的是程序启动时的开销(对于执行时间很短的程序这一点尤为重要),还是稳定运行之后的开销。对后者而言,实验者需要关注系统的热身效应,例如类系加载或动态代码优化。不论哪种情况,都应当排除冷启动效应的影响,例如向磁盘缓存上加载必要文件导致的延迟。正是因为这些原因, Georges 等 [2007] 主张多运行几次虚拟机以及基准测试程序实例,并且排除第一次的执行结果。

动态(或者运行时)编译是不确定因素(non-determinism)的一个主要来源,在对不同的回收算法进行比较时这一因素很难处理。一种解决方案是排除这一因素:研究者可以通过编译重放(compiler replay),记录哪些方法经过了优化,以及基准测试程序做了何种级别的准备,这一记录可以确保虚拟机在接下来的性能测试过程中使用同一级别的优化 [Blackburn 等, 2006]。但这一方法存在的问题是,不同的算法实现通常会调用不同的方法,特别是在被测试的组件中,因此使用哪个编译记录经常使人困惑,是二选一?还是取其交集?

实验结果在任何情况下都应当是有效的,即使可能存在一些偏差(例如一些随机误差),这需要多次进行实验并对其结果进行统计学比较。如果要确信一种方法比另一种方法优秀,首先必须描述其可信度,其次每个备选项的可信区间应当是从结果中得出的,并且这些区间之间没有重叠。Georges 等 [2007] 提出了一种严格的统计学方法来应对不确定性以及不可预知的误差(包括动态编译所带来的误差)。他们主张调用虚拟机的一个实例,并多次执行一个基准程序直到稳定状态(即最后 k 次基准迭代的变化系数[⊖]小于某一预定的阈值)。通过这 k 次迭代便可计算出稳定状态下某一基准应用程序在某一方面的平均值。重复这一过程即可计算出整体平均值以及可信区间,更进一步便可得出整体分布(或者至少是某些瞬间的分布)。

垃圾回收的研究需要充分的性能报告。对于简单的点阵图,即使辅它以可信区间,也是远远不够的,因为内存管理涉及时间和空间两方面的权衡。大多数环境下,降低回收停顿时间的一种方法是增大堆的空间(增大到一定程度后将达到最优,但如果再增大,则由于局部

⊖ 变化系数即标准差除以平均值。

性原理，执行时间将会变长)。如果仅基于一种大小的堆进行实验，实验结果将不足信，因此实验者拥有调整堆大小(以及堆内部的子空间大小)的能力是十分重要的，只有这样才能对一种内存管理算法进行全面的性能评估。对于那些可以通过自动改变堆大小来优化性能的生产级虚拟机，我们也持相同的观点，因为其自适应能力可能只是针对最终用户的，而研究者和开发者可能需要关注更多的内部细节。

垃圾回收系统内部的复杂性更加凸显了充分进行性能实验的重要性。这里的复杂性表现在：即使对回收器参数进行普通的微调也会导致回收器的行为发生很大变化，例如在回收器的调度方面，即使对回收过程的发生时间做很小的调整，也会使一个较大数据结构的命运发生变化(是依旧可达，还是成为垃圾)，不论是对于本次回收过程的开销还是下一次回收的发生时间，这都将产生很大影响，因此这一参数具有自我放大能力。如果基于不同大小的堆(通常是程序正常执行所需要的最小堆大小的整数倍)进行实验，那么实验结果的抖动将是显而易见的。

1.6 术语和符号

在接下来的一节中我们将描述本书中用到的各种符号。对于前文中提到的部分术语，我们也将给出更为精确的定义。

首先需要说明的是存储的单位。我们遵循一个字节包含八个位这一惯例。我们简单地使用 KB (kilobyte)、MB (megabyte)、GB (gigabyte)、TB (terabyte) 来描述对应的 2 的整数次幂内存单元(分别是 2^{10} 、 2^{20} 、 2^{30} 、 2^{40})，而不使用 SI 数字前缀的标准定义。

1.6.1 堆

堆是由一段或者几段连续内存组成的空间集合。内存颗粒 (granule) 是堆内存分配的最小单位，一般是一个字 (word) 或者一个双字 (double-word)，这取决于需要的对齐 (alignment) 方式。内存块 (chunk) 是一组较大的连续内存颗粒。内存单元 (cell) 是由数个连续颗粒组成小内存块，通常用于内存的分配和释放，也可能由于某种原因被浪费或闲置。

对象 (object) 是为应用程序分配的内存单元。对象通常是一段可寻址的连续字节或字的数组，其内部被划分成多个槽 (slot) 或者域 (field)，如图 1.3 所示(尽管在某些实时系统或者嵌入式系统中的内存管理器会通过指针结构来实现较大的独立对象，但是这一结构并不暴露给用户程序)。域可能会包含一些非引用的纯数据，例如整数。引用可以是指向某个堆中对象的指针，也可以是空 (NULL)。引用通常是一个正规指针 (canonical pointer)，指向对象头部(即对象首地址)或者距头部有一定偏移量的地址。某些情况下，对象会用一个头域 (header field) 来存放运行时系统会用到的元数据，它一般(但并不绝对)位于对象的起始地址。派生指针 (derived pointer) 一般是在对象的正规指针的基础上增加一个偏移量而得到的指针，内部指针 (interior pointer) 是指向内部对象域的派生指针。

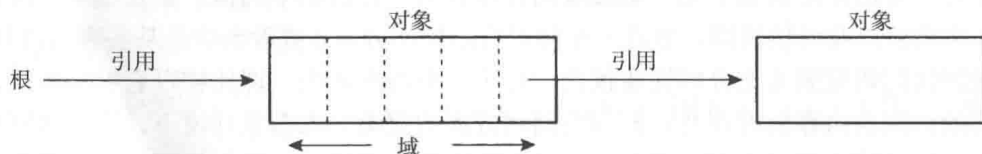


图 1.3 根、堆内存单元以及引用。对象(以长方形表示)通常可以划分成多个域(以虚线划分)，实线箭头表示引用

内存块 (block) 是依照特定大小 (通常是 2 的整数次幂) 对齐的大块内存。作为补充, 这里引入帧 (frame) 和空间 (space) 的概念。帧 (与“栈帧”的概念无关) 是地址空间中一大段 2^k 大小的地址空间, 空间是由一系列 (可能) 不连续的内存块 (甚至对象) 组成的集合, 且系统处理每个内存块的方式相似。页 (page) 是由硬件以及操作系统的虚拟内存机制定义的, 高速缓存行 (cache line) 或者高速缓存块 (cache block) 是由 CPU 的高速缓存 (cache) 定义的。卡 (card) 是以 2 的整数次幂对齐的内存块, 其大小一般小于一页, 且通常与某些跨空间指针 (见 11.8 节) 的方案有关。

堆通常可以使用对象图 (object graph) 的方式来描述, 它一般是一个有向图 (directed graph), 图的节点 (node) 是堆中的对象, 有向边是对象之间的引用。边一般是从源节点 (source node) 或者根 (root) (见下文) 指向目标节点 (destination node) 的引用。

1.6.2 赋值器与回收器

对于使用垃圾回收的程序, Dijkstra 等 [1976、1978] 将其执行过程划分为两个半独立的部分:

- 赋值器执行应用代码。这一过程会分配新的对象, 并且修改对象之间的引用关系, 进而改变堆中对象图的拓扑结构, 引用域可能是堆中对象, 也可能是根, 例如静态变量、线程栈等。随着引用关系的不断变更, 部分对象会失去与根的联系, 即从根出发沿着对象图的任何一条边进行遍历都无法到达该对象。
- 回收器 (collector) 执行垃圾回收代码, 即找到不可达对象并将其回收。

一个程序可能拥有多个赋值器线程, 但是它们共用同一个堆。相应的, 也可能存在多个回收器线程。

1.6.3 赋值器根

与堆内存不同, 赋值器的根是一个有限的指针集合, 赋值器可以不经过其他对象直接访问这些指针。堆中直接由赋值器根所引用的对象称为根对象 (root object)。当赋值器访问堆中的对象时, 它需要从当前的根对象集合中加载指针 (进而会增加新的根)。赋值器也可能会丢弃一些根, 例如将某个根指针改写为新的引用 (即改写为空指针或者指向其他对象的指针)。我们用 `Roots` 来表示根集合。

[12]

在实际情况下, 根通常包括静态 / 全局存储以及线程本地存储 (例如线程栈), 赋值器可以通过根中的指针直接操纵堆中对象。在赋值器线程执行一段时间后, 线程状态 (及其根集合) 可能发生变化。

在类型安全的语言中, 一旦某个对象在堆中不可达, 并且赋值器的所有根指针中也不包含对该对象的引用, 赋值器将无法再次访问该对象。赋值器 (在没有与运行时系统交互的情况下) 不能随意地“重新发现”该对象, 因为其不能通过任何指针到达该对象, 也不能在该对象的地址上构造新对象。在某些语言中, 部分对象需要进行终结 (finalisation), 即当这些对象不可达时, 运行时系统会将其复活, 从而赋值器会再次访问到复活之后的对象。这里我们关注的是, 赋值器在没有外部系统协助的情况下是无法访问到不可达对象的。

1.6.4 引用、域和地址

堆中对象的引用通常是基于它的内存地址来实现的 (该地址可以是对象的首地址, 也可

以是指向对象内部某个数据或者元数据的标准指针)。对于任意给定对象(地址) N , 不论其内部是否包含指针, 我们都可以将其视为一个域数组, 并通过该数组来访问对象的内部域: 从 0 开始, 对象的第 i 个域可以用 $N[i]$ 来表示, 域的总数可以写作 $|N|$ 。我们使用 C 风格的语法来描述解引用(dereference)操作, 例如对指针 p 的解引用可以写作 $*p$; 同理, 使用 $\&$ 符号来获取对象的地址, 因此可以用 $\&N[i]$ 表示对象 N 中第 i 个域的地址。对于给定对象(地址) N , $\text{Pointers}(N)$ 表示 N 中指针域的集合, 更加正式的写法是:

$$\text{Pointers}(N) = \{a \mid a = \&N[i], \forall i : 0 \leq i < |N| \text{ 若 } N[i] \text{ 是一个指针} \}$$

为了方便表述, 我们使用 Pointers 来表示堆中所有对象包含的所有指针域, 相应的, Nodes 表示堆中所有(已分配)对象的集合。我们也把 Roots 当作(与堆相隔离的)伪对象, 同时定义 $\text{Pointers}(\text{Roots}) = \text{Roots}$, 从而可以用 $\text{Roots}[i]$ 代表第 i 个根域。

1.6.5 存活性、正确性以及可达性

如果某一对象在程序的后续执行过程中可能会被赋值器访问, 则称该对象是存活(live)的。回收器的正确性是指其永远不会回收依然存活的对象, 但对于应用程序而言, 存活性(liveness)是一个不确定的特征: 一般的程序无法确定赋值器是否永远不会再访问某个堆中对象^①, 因为程序持有一个对象的指针并不意味着会对其进行访问。幸运的是, 我们可以将指针可达性(pointer reachability)这个可确定因素作为对象是否存活的近似等价, 即: 如果从对象 M 的域 f 出发, 经过一条指针链最终可以到达对象 N , 则称对象 N 从对象 M 可达。因此如果从赋值器根出发, 经过一条指针链可以到达某一对象, 赋值器才有可能访问到该对象。

更加正式的(在数学意义中进行可达性表述), 我们使用 \rightarrow_f 表示直接指针可达, 并将其定义如下: 对于堆中任意两个节点 M 和 N , 当且仅当 $\text{Pointers}(M)$ 中存在一个域 $f = \&M[i]$, 且 $*f = N$ 时, 才有 $M \rightarrow_f N$ 。同理, 当且仅当根集合中存在域 f , 且 $*f = N$ 时, 才有 $\text{Roots} \rightarrow_f N$ 。如果 $\text{Pointers}(M)$ 中存在域 f , 使得 $M \rightarrow_f N$ (也就是 M 的域 f 指向 N), 则可以说对象 N 从对象 M 直接可达, 记作 $M \rightarrow N$ 。因此堆中可达对象的集合是根集合在“ \rightarrow ”关系下的递归引用闭包, 即:

$$\text{reachable} = \{N \in \text{Nodes} \mid (\exists r \in \text{Roots} : r \rightarrow N) \vee (\exists M \in \text{reachable} : M \rightarrow N)\} \quad (1.1)$$

对于在堆中不可达, 同时也不被任何赋值器根引用的对象, 其不可能被一个类型安全的赋值器访问到, 也就是说, 赋值器能访问到的只可能是堆中的可达对象, 因此垃圾回收器可以使用可达性来定义存活性。不可达对象必然是死亡对象, 因此可以将其回收, 可达对象可能仍然存活, 因此不能将其回收。我们倾向于在概念上将存活与可达等价、将死亡与不可达等价、将垃圾与不可达对象等价, 尽管这在严格意义上并不准确。

1.6.6 伪代码

我们使用常见的伪代码来描述垃圾回收算法。我们提供的算法片段仅仅是说明性的而非定义性的, 相对于较为正式的伪代码, 我们宁愿通过非正式的文本来解决歧义。我们的目标是简洁且有代表性地对每种算法进行描述, 而不是全方位地描述其具体实现。

我们使用缩进来表示伪代码段以及控制语句的范围。用符号“ \leftarrow ”表示赋值操作, 符号

^① 存活性的不可判定性是停机问题(halting problem)的一个推论(所谓停机问题, 即判定程序是否会在有限时间内运行结束的问题)。——译者注

“=”表示相等，其他逻辑操作符与关系操作均使用C风格符号，例如“||”(逻辑或)、&&(逻辑与)、≤、≥、≠、%(模)等。

1.6.7 分配器

分配器 (allocator) 与回收器在功能上是正交关系。分配器支持两种操作：分配 (allocate) 和释放 (free)。分配是为某一对象保留底层的内存存储，释放是将内存归还给分配器以便复用。分配存储空间的大小是由一个可选参数来控制的，如果我们在伪代码中忽略这一参数，意味着分配器将返回一个固定大小的对象，或者对象大小对于算法的理解并非必要。分配操作也可能支持更多参数，例如将数组的分配与单个对象的分配进行区分，或者将指针数组的分配和不包含指针的数组进行区分，或者包含其他一些必要信息以便初始化对象头部。

1.6.8 赋值器的读写操作

赋值器线程在工作过程中会执行三种与回收器相关的操作：创建 (New)、读 (Read)、写 (Write)。我们约定，赋值器操作的命名均采用首字母大写的方式，与回收器相关的操作则均采用首字母小写。这些操作通常都会有顾名思义的行为：分配一个新对象、读一个对象的域、写一个对象的域。某些特殊的内存管理器会为基本操作增加一些额外功能，即屏障 (barrier)，屏障操作会同步或者异步地与回收器产生交互。在后文，我们将区分读屏障 (read barrier) 和写屏障 (write barrier)。

14

New() New 操作从堆分配器中获得一个新的堆对象，分配器还会返回新分配对象的首地址。堆内存分配的实现方式有很多种，但在新分配的对象可被赋值器操作之前，回收器都需要先对其元数据进行初始化。在默认情况下，New 操作的平凡定义是简单地执行内存分配操作。

```
New():
    return allocate()
```

Read(src, i) Read 操作访问内存中的某一对象的域，并且返回该域所记录的值，其中保存的既可以是纯值 (scalar) 也可以是指针。Read 操作会引发内存加载操作，它需要两个参数：指向对象的指针、待访问的域的索引号。如果 src 中的域 src[i] 是根 (即 &src[i] ∈ Roots)，则认为 src = Roots。在默认情况下，Read 的平凡定义是简单地返回域的内容。

```
Read(src, i):
    return src[i]
```

Write(src, i, val) Write 操作改变特定内存的值。该操作引发内存存储，它需要三个参数：指向源对象的指针、待修改域的索引号、待存储的值 (纯值或者指针)。与 Read 操作相同，如果 src 中的域 src[i] 是根 (即 &src[i] ∈ Roots)，则认为 src = Roots。在默认情况下，Write 操作的平凡定义是简单地更新域。

```
Write(src, i, val):
    src[i] ← val
```

1.6.9 原子操作

在面对赋值器线程之间、回收器线程之间、赋值器和回收器之间的并发操作时，所有的

垃圾回收算法都需要一些“看起来”可以原子执行的代码序列。例如，挂起赋值器线程可以确保垃圾回收过程的原子性（atomically），即在垃圾回收过程中赋值器不会访问堆中对象；更进一步，当回收器与赋值器并发工作时，对于回收器和赋值器线程而言，创建、读、写操作必须“看起来”是原子性的。为了简化垃圾回收算法的描述，我们一般不会去深究这些原子操作的具体实现，只是简单地用“atomic”这个关键字来描述它们。原子操作的所有步骤看起来都应当是不可分割的、瞬时的，也就是说，其他操作只能在原子操作之前或者之后执行，但是不能在原子操作内部的任意两个步骤之间执行。我们将在第 11 章和第 13 章对原子操作的不同实现技术进行讨论。

1.6.10 集合、多集合、序列以及元组

一般使用抽象数据结构来描述算法。我们将适当地使用数学符号来描述简单概念，并避免过于晦涩的数学符号。大多数情况下，我们关注的是集合（set）与元组（tuple），它们最基本的操作是元素的增、删、查。

15

我们将集合定义为存放不同元素的容器（其中的元素两两不同）。集合中元素的数量称为基（cardinality），写作 $|S|$ 。

除了标准的集合符号，我们也使用多集合（multiset），其内部允许存在相同元素。多集合的基是其中元素的总数量，包括相同的元素。某一元素出现的次数称为该元素的重数（multiplicity）。多集合的相关符号如下：

- $[]$ 表示空的多集合。
- $[a, a, b]$ 表示多集合中包含两个 a 以及一个 b 。
- $[a, b] + [a] = [a, a, b]$ 表示多集合的求并操作。
- $[a, a, b] - [a] = [a, b]$ 表示多集合的相减操作。

序列（sequence）是一组元素的有序列表。与集合（或者多集合）不同，序列中有顺序的概念。类似于多集合，相同的元素可以在一个序列的不同位置多次出现。序列的相关符号如下：

- $()$ 表示空序列。
- (a, a, b) 表示序列中包含两个 a 和一个 b 。
- $(a, b) \cdot (a) = (a, b, a)$ 表示将序列 (a) 追加到序列 (a, b) 之后。

长度为 k 的元组（tuple）与相同长度的序列等价，但是其长度一般不允许变化，因此我们一般采用一个不同的符号来描述元组。一般情况下，元组包含两个或者更多个元素。元组相关的符号如下：

16

- $\langle a_1, \dots, a_k \rangle$ 表示一个长度为 k 的元组，其第 i 个元素是 a_i ，其中 $1 \leq i \leq k$ 。

标记 - 清扫回收

标记 - 清扫 (mark-sweep)、标记 - 复制 (mark-copy)、标记 - 整理 (mark-compact)、引用计数 (reference counting) 是 4 种最基本的垃圾回收策略。大多数回收器可能会以不同的方式对这些基本回收策略进行组合, 例如在堆中某一区域使用一种回收策略, 而在另一个区域使用另一种回收策略。接下来的 4 章将专注于这 4 种基本回收策略的介绍, 第 6 章将对它们的特点进行比较。

在对这 4 种基本回收策略进行描述时, 我们假设赋值器运行在一个或者多个线程之上, 且只有一个回收器线程, 当回收器线程运行时, 所有的赋值器线程均处于停止状态。这种“万物静止” (stop the world) 的策略大幅简化了回收器的实现。从赋值器线程角度来看, 回收过程的执行是原子性的, 即赋值器线程感知不到回收器的任何中间状态, 回收器也不会受到赋值器线程的任何干扰。我们假设当回收过程开始时, 赋值器线程停止在一个允许回收器安全扫描其根集合的回收点, 具体的运行时接口我们将在第 11 章详述。通过“万物静止”的方法可以获取堆的快照, 因此回收器在进行对象存活性判定时不用担心赋值器将堆中对象的拓扑结构重新排列。这同时也意味着在回收器释放内存的过程中, 我们无需担心在同一时刻存在其他回收器释放内存或者赋值器线程申请内存, 进而避免在线程之间引入额外的同步机制。多赋值器线程同时获取内存的情况我们将在第 7 章讨论, 多回收器线程, 或者赋值器与回收器并发执行的情况将导致运行时系统更加复杂, 我们将在后面的章节中进行讨论。

我们建议读者先熟悉接下来 4 章中介绍的基本回收算法, 然后再进一步了解后面章节中提到的更加高级的回收器。有经验的读者可能会跳过这些基础算法的描述, 但是增加对这些基础回收器实现方式的了解也不失为一件有趣的事。认为第 2 ~ 6 章的部分内容过于精简的读者, 可以参考 Jones[1996] 中的资料, 其对经典算法的描述更加详细, 并且包含更多的示例。

理想的垃圾回收的目的是回收程序不再使用的对象所占用的空间, 任何自动内存管理系统都面临 3 个任务:

- 1) 为新对象分配空间。
- 2) 确定存活对象。
- 3) 回收死亡对象所占用的空间。

这些任务并非相互独立, 特别是回收空间的方法影响着分配新空间的方法。正如第 1 章所提到的, 真正的存活性问题是一个不可确定的问题, 因此我们使用指针可达性 (参见 1.6 节) 来近似对象的存活性: 只有当堆中存在一条从根出发的指针链能最终到达某个对象时, 才能认定该对象存活, 更进一步, 如果不存在这样一条指针链, 则认为对象死亡, 其空间可以得到回收。尽管存活对象集合中可能包含一些永远不会再被赋值器访问的对象, 但是死亡对象集合中的对象却必定是死亡的。

标记 - 清扫算法 [McCarthy, 1960] 是我们将要介绍的第一种回收算法, 它是在指针可达性递归定义指导下最直接的回收方案。它的回收过程分为两个阶段: 第一阶段为追踪

(trace) 阶段, 即回收器从根集合 (寄存器、线程栈、全局变量) 开始遍历对象图, 并标记 (mark) 所遇到的每个对象; 第二阶段为清扫 (sweep) 阶段, 即回收器检查堆中每一个对象, 并将所有未标记的对象当作垃圾进行回收。

标记-清扫算法是一种间接回收 (indirect collection) 算法, 它并非直接检测垃圾本身, 而是先确定所有存活对象, 然后反过来判定其他对象都是垃圾。需要注意的是, 该算法的每次调用都需要重新计算存活对象集合, 但并非所有的垃圾回收算法都需要如此。第5章将介绍一种直接回收 (direct collection) 策略, 即引用计数。与间接回收不同, 直接回收可以通过对象本身来判断其存活性, 因此其不需要额外的追踪过程。

2.1 标记-清扫算法

从垃圾回收器角度来看, 赋值器线程所执行的只有3种操作: 创建、读、写。每一个回收算法必须对这3种操作进行合理的重定义 (1.6节给出了默认的定义)。标记-清扫算法与赋值器之间的接口十分简单: 如果线程无法分配对象, 则唤起回收器, 然后再次尝试分配 (如算法2.1所示)。为了强调回收器工作在“万物静止”模式下而非与赋值器线程并发执行, 我们特别使用 `atomic` 关键字来标记回收程序。如果回收完成之后仍然没有足够的内存以满足分配需求, 则说明堆内存耗尽, 这通常是一个严重的错误。某些语言在遇到这一情况时会抛出异常, 开发者可以将其捕获, 如果可以通过删除引用的方式释放内存 (例如释放那些在未来可以重新创建的数据结构), 那么可以再次请求分配。

算法 2.1 标记-清扫: 分配

```

1 New():
2     ref ← allocate()
3     if ref = null                                /* 堆中无可用空间 */
4         collect()
5         ref ← allocate()
6         if ref = null                            /* 堆中仍然无可用空间 */
7             error "Out of memory"
8     return ref
9
10 atomic collect():
11     markFromRoots()
12     sweep(HeapStart, HeapEnd)

```

回收器在遍历对象图之前必须先构造标记过程需要用到的起始工作列表 (work list) (即算法2.2中的 `markFromRoots` 方法), 即对每个根对象进行标记并将其加入工作列表 (第11章将讨论如何寻找根集合)。回收器可以通过设置对象头部某个位 (或者字节) 的方式对其进行标记, 该位 (字节) 也可位于一张额外的表中。不包含指针的对象不会有任何后代, 因此无需将其加入工作列表, 但仍需将其标记。为了减少工作列表的大小, `markFromRoots` 方法在将线程根加入到工作列表之后立刻调用 `mark` 方法, 而如果将 `mark` 方法 (算法2.2的第8行) 从循环中提出则可加快线程根扫描, 例如并发回收器可能只需要挂起线程并扫描其栈, 而接下来的对象图遍历则可以和赋值器并发进行。

算法 2.2 标记-清扫: 标记

```

1 markFromRoots():
2     initialise(worklist)

```

```

3   for each fld in Roots
4       ref ← *fld
5       if ref ≠ null && not isMarked(ref)
6           setMarked(ref)
7           add(worklist, ref)
8           mark()
9
10  initialise(worklist):
11      worklist ← empty
12
13  mark():
14      while not isEmpty(worklist)
15          ref ← remove(worklist)          /* ref 已经标记过 */
16          for each fld in Pointers(ref)
17              child ← *fld
18              if child ≠ null && not isMarked(child)
19                  setMarked(child)
20                  add(worklist, child)

```

单线程回收器可以基于栈来实现工作列表，此时回收器将以深度优先顺序遍历（depth-first traversal）对象图。如果将标记位保存在对象中，那么 mark 方法所处理的将是那些刚刚被标记的对象，因此这些对象很可能仍在硬件高速缓存中。正如我们反复提到的，回收过程的高速缓存相关行为会影响到回收器的性能，我们将在稍后讨论提升回收器局部性的技术。

标记存活对象的过程非常直观：先从工作列表中获取一个对象的引用，然后对其所引用的其他对象进行标记，直到工作列表变空为止。需要注意的是，在这一版本的 mark 方法中，工作列表中的每个对象都拥有自身的标记位。如果某一指针域的空，或者其指向的对象已经标记过，则无需对其进行处理，否则回收器需要对其目标对象进行标记并将其添加到工作列表中。

标记阶段完成的标志是工作列表变空，而不是将所有已标记对象都添加到工作列表。此时回收器已经完成每个可达对象的访问与标记，任何没有打上标记位的对象都是垃圾。

19

在清扫阶段，回收器会将所有未标记的对象返还给分配器（如算法 2.3 所示）。在这一过程中，回收器通常会在堆中进行线性扫描，即从堆底开始释放未标记的对象，同时清空存活对象的标记位以便下次回收过程复用。另外，如果标记过程使用两个标记位，且连续两次标记过程使用的标记位不同，则可以省去清空标记位的开销。

算法 2.3 标记 - 清扫：清扫

```

1  sweep(start, end):
2      scan ← start
3      while scan < end
4          if isMarked(scan)
5              unsetMarked(scan)
6          else free(scan)
7          scan ← nextObject(scan)

```

我们将在第 7 章之后讨论 allocate 和 free 的具体实现细节，但需要注意的是，标记 - 清扫回收器要求堆布局满足一定的条件：第一，标记 - 清扫回收器不会移动对象，因此内存管理器必须能够控制堆内存碎片，这是因为过多的内存碎片可能会导致分配器无法满足新分配请求，从而增加垃圾回收频率，在更坏情况下，新对象的分配可能根本无法完成；第二，清扫

器必须能够遍历堆中每一个对象，即对于给定对象，不管其后是否存在一些用于对齐的填充字节，sweep 方法都必须能够找到下一个对象，因此用 nextObject 方法要完成堆的遍历，仅获取对象的大小信息是远远不够的（算法 2.3 中的第 7 行）。我们将在第 7 章讨论堆的可遍历性。

2.2 三色抽象

三色抽象 (tricolour abstraction) [Dijkstra 等, 1976, 1978] 可以简洁地描述回收过程中对象状态的变化（是否已被标记、是否在工作列表中等）。三色抽象是描述追踪式回收器的一种十分有用的方法，利用它可以推演回收器的正确性，这正是回收器必须保证的。在三色抽象中，回收器将对象图划分为黑色对象（确定存活）和白色对象（可能死亡）。任意对象在初始状态下均为白色，当回收器初次扫描到某一对象时将其着为灰色，当完成该对象的扫描并找到其所有子节点之后，回收器会将其着为黑色。从概念上讲，黑色意味着已经被回收器处理过，灰色意味着已经被回收器遍历但尚未完成处理（或者需要再次进行处理）。三色抽象也可以推广到对象的域中：灰色表示正在处理的域，黑色表示已经处理过的域。如果把赋值器也当作一个对象，则三色抽象也可用于推演赋值器根集合的状态变化 [Pirinen, 1998]：灰色赋值器表示回收器尚未完成对其根集合的扫描，黑色赋值器表示回收器已经完成对其根集合的扫描（并且不需要再次扫描）^①。一次堆遍历过程可以形象地看作是回收器以灰色对象作为“波面”（wavefront），将黑色对象和白色对象分离，不断向前推进波面，直到所有可达对象都变成黑色的过程。

20

在标记-清扫回收过程中，对象的颜色变化过程如图 2.1 所示。图 2.1 展示了一个基本的对象图以及标记栈（即工作列表的具体实现）在标记过程中某个阶段的状态。标记栈中的所有对象都会再次得到访问，因此它们是灰色的，所有已经标记但不在标记栈中的对象都是黑色的（如图 2.1 中对象图的根），其他对象均为白色（当前的对象 A、B、C）。但是，一旦 mark 方法完成对图的遍历，标记栈将变为空（即没有灰色对象），则只有对象 C 依然会是白色（垃圾），其他对象都将得到标记（黑色）。

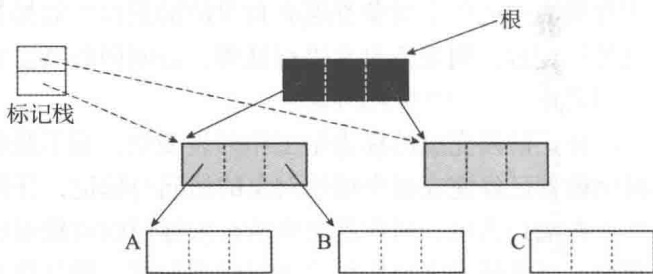


图 2.1 使用三色抽象来演示标记过程。黑色表示回收器已经将该对象及其后代处理完成，灰色表示回收器已经扫描到对象但尚未完成处理，白色表示回收器尚未扫描到对象（某些对象可能永远无法扫描到）

上述算法中存在一个重要的不变式：在标记过程完成后，对象图中将不可能存在从黑色对象指向白色对象的引用，因此在标记过程中，所有白色可达对象都只能是从灰色对象可达。如果这一不变式被打破，那么回收器将不会进一步处理黑色对象，从而可能导致某个黑色对象的后代可达但未被标记（进而被错误地释放）。后面我们将看到，在研究赋值器线程与回收器线程并发执行的并发垃圾回收时，垃圾回收状态三色视图将十分有用。

2.3 改进的标记-清扫算法

程序的性能通常与其高速缓存的相关行为有很大关系。从内存中加载一个值可能要花费

^① 灰色赋值器与黑色赋值器的概念会在并发回收算法中用到，参见第 15.1 节。——译者注

上百个时钟周期，但从 L1 高速缓存 (L1 cache) 中加载可能只需要花费三到四个时钟周期。高速缓存之所以能够提升程序的性能，主要是因为程序在运行时表现出了良好的时间局部性 (temporal locality)，即一旦程序访问了某个内存地址，则很可能在不久之后再次访问该地址，因此值得将它的值缓存。程序也可能表现出良好的空间局部性 (space locality)，即一旦程序访问了某个内存地址，则很有可能在不久之后访问该地址附近的数据。现代硬件可以从两个方面利用程序的局部性特征：一方面，高速缓存与更低级别内存之间不会进行单个字节的数据传输，而是以一个固定的字节数为最小传输单元 (即高速缓存行或者高速缓存块的大小)，通常是 32 ~ 128 字节；另一方面，处理器可能会使用硬件预取 (prefetch) 技术，例如 Intel Core 微处理器架构可以探测到有规律的步进内存访问操作，进而提前读取数据流。开发者也可以利用显式预取指令引导预取过程。

21

不幸的是，垃圾回收器的行为与典型应用程序并不相同，标记 - 清扫回收器的时间局部性并不明显的。尽管程序中可能存在一些被引用次数非常多的对象，但大部分对象都不是共享的 (即只被一个指针所引用)，因而回收器在标记阶段通常只会读写对象头部各一次 [Printezis and Grathwaite, 2002]，即：回收器读取对象头部的标记位，如果该对象尚未得到标记则设置其标记位，除此之外回收器在标记阶段一般不会再次访问该对象。对象头部通常会包含一个指向该对象类型信息的指针 (对象类型信息可能也是一个对象)，回收器据此来获取对象中的指针域。这类型信息可能包含指针域的描述信息，也可能包含将对象自身标记并将其子节点添加到标记栈的代码。程序通常只使用有限的几种类型，并且个别类型的使用频率要远远大于其他类型，因此对类型信息进行缓存的价值较高。对于将标记位放置在对象头部的策略，由于堆中对象在标记阶段往往只会被访问一次，所以硬件预取无法发挥功效。

接下来，我们将探讨优化标记 - 清扫回收器性能的几种方法。

2.4 位图标记

回收器可以将对象的标记位保存在其头部的某个字中，除此之外也可以使用一个独立的位图来维护标记位，即：位图中的每个位关联堆中每个可能分配对象的地址。位图所需的空间取决于虚拟机的字节对齐要求。位图可以只有一个，也可以存在多个，例如在块结构的堆中，回收器可以为每个内存块维护独立的位图，这一方式可以避免由于堆不连续导致的内存浪费。回收器可以将每个内存块的位图置于其自身内部，但如果所有内存块中位图的相对位置全部相同，则可能导致性能的下降，因为不同内存块的位图之间可能会争用相同的组相关高速缓存 (set-associative cache)。对位图的访问同时也意味着对位图所在页的访问 (即可能导致缺页异常——译者注)，因此基于换页和高速缓存相关性的考虑，在访问位图时花费更多的指令以保持程序的局部性通常来说是值得的。为避免高速缓存的相关问题，可以将内存块中位图的位置增加一个简单偏移量，例如内存块地址的简单哈希值。还可以将位图存放在一个额外的区域 [Boehm and Weiser, 1988 年] 中，并以其所对应内存块的哈希值等作为索引，这样既避免了换页问题，也避免了高速缓存冲突。

位图标记通常仅适用于单线程环境，因为多线程同时修改位图可能存在较大的写冲突风险。设置对象头部中的标记位通常是安全的，因为该操作是幂等的，即最多只会将标记位设置多次。相对于位图，实践中更常用的是字节图 (byte-map)，虽然它占用的空间是前者的 8 倍，但却解决了写冲突问题。另外还可以使用同步操作来设置位图中的位。在实际应用中，

如果将标记位保存在对象头部通常会带来额外的复杂度，因为头部通常会存放一些赋值器共享数据，例如锁或者哈希值，那么当标记线程与赋值器线程并发执行时可能会产生冲突。因此，为了确保安全，标记位通常会占用头部中一个额外的字，以便与赋值器共享数据区分，当然也可以使用原子操作来设置头部中的标记位。

22

使用标记位图具有诸多潜在优点，我们将进行逐一描述，并检验这些优点在现代硬件条件下是否可以得到充分发挥。相对于将标记位放置在对象头部这一策略，位图可以使得标记位更加密集；对于使用位图的标记-清扫回收器，标记过程只需读取存活对象的指针域而不会修改任何对象；对于不包含引用的对象，回收器只需要读取其类型信息描述域；清扫器不会对存活对象进行任何读写操作，它只会在释放垃圾对象的过程中覆盖其某些域（例如将它们链接到空闲链表上）。因此，位图标记不仅可以减少内存中需要修改的字节数，而且减少了对高速缓存行的写入，进而减少需要写回内存的数据量。

位图标记最初应用在保守式回收器（conservative collector）中。保守式回收器的设计初衷是为 C 和 C++ 等“不合作”的语言提供自动内存管理功能 [Boehm and Weiser, 1988]。类型精确（type-accurate）系统可以精确地识别每一个包含指针的槽，不论其位于对象中，还是位于线程栈或者其他根集合中，而保守式回收器则无法得到编译器和运行时系统的支持，因而其在识别指针时必须采用保守的判定方式，即：如果槽中某个值看起来像是指针引用，那么就必须假定它是一个指针。我们将在第 11 章详细讨论指针识别问题。保守式回收器可能错误地将一个槽当作指针，这带来了两个安全上的要求：第一，回收器不能修改任何赋值器可能访问到的内存地址的值（包括对象和根集合）。这一要求导致保守式回收器不能使用任何可能移动对象的算法，因为对象被移动之后需要更新指向该对象的所有引用。这同时也导致在头域中保存标记位的方案不可行，因为错误的指针会指向一个实际并不存在的“对象”，因此设置或者清理标记位可能会破坏用户数据。第二，应当尽可能减少赋值器破坏回收器数据的可能性。与将标记位等回收器元数据存放在一个单独区域的方案相比，为每个对象增加一个回收器专用头部数据会存在更高的风险。

使用位图标记的另一个重要目的是减少回收过程中的换页次数 [Boehm, 2000]。在现代系统中，任何由回收器导致的换页行为通常都是不可接受的，因此位图标记是否可以提升高速缓存性能便成为一个值得关注的问题。许多证据表明，对象往往成簇诞生并成批死亡 [Hayes, 1991; Jones, Ryder, 2008]，而许多分配器往往也会将这些对象分配在相邻的空间。使用位图来引导清扫可以带来两个好处：第一，在位图/字节图中，一个字内部的每个位/字节全部都被设置/清空的情况会经常出现，因此回收器可以批量读取/清空一批对象的标记位；第二，通过位图标记可以更简单地判定某一内存块中的所有对象是否都是垃圾，进而可能一次性回收整个内存块。

许多内存管理器都使用块结构堆（如 Boehm and Weiser[1988]）。最直接的位图标记实现策略可能是在每个内存块的前端保留一块内存以用作位图。但正如我们前面所提到的，这一策略可能会导致不必要的高速缓存冲突或者换页，因此回收器通常将位图与用户数据块分开并单独存放。

Garner 等 [2007] 提出了一种混合标记策略，即将分区适应分配器（segregated fits allocator）所管理的每个数据块与字节图中的一个字节相关联，同时依然保留对象头部的标记位。当且仅当内存块中至少存在一个存活对象时，该内存块所对应的标记字节才会被设置。清扫器可以根据字节图快速地判定某一内存块是否完全为空（即不包含存活对象），进

而可以将其整体回收。这一策略有两个优点：第一，在并发情况下，无需使用同步操作来设置字节图中的标记字节以及对象头部的标记位；第二，写操作没有数据依赖（这可能导致高速缓存延迟），且对字节图中标记字节的写操作也是无条件的。

Printezis 和 Detlefs[2000] 在一个主体并发分代回收器（mostly-concurrent, generational collector）中使用位图来减少标记栈所占用的空间。与其他方法类似，回收器首先在位图中对赋值器的根进行标记，然后标记线程通过线性扫描位图来寻找存活对象。在算法 2.4 中，回收器将遵从如下不变式：位于当前指针（即 mark 方法中的指针 cur）之后的所有已标记对象均为黑色，而位于当前指针之前的所有已标记对象均为灰色。根据“将对象从栈中弹出，并且递归地标记其后代，直到栈为空”的原则，当回收器找到下一个已标记存活对象 cur 时会将其压入标记栈中，并继续进行循环。在 markStep 中，如果新追踪到的子节点地址小于堆中的 cur，则回收器将其压入标记栈，否则该对象的处理将被推迟到后续的线性查找过程中。该算法与算法 2.1 的主要区别在于将对象子节点压入标记栈的条件，即算法 2.4 中的第 15 行。在算法 2.4 中，回收器的黑色波面将在堆中线性移动，只有位于该波面之后的对象才会得到标记（即被压入标记栈）。尽管该算法的时间复杂度与待回收空间的大小成正比，但在实际应用中，位图查找的开销通常较低。

23

算法 2.4 Printerzis 和 Detlefs 的位图标记

```

1 mark()
2   cur ← nextInBitmap()
3   while cur < HeapEnd /* 当且仅当 ref < cur 时，已标记对象 ref 才是黑色的 */
4       add(worklist, cur)
5       markStep(cur)
6       cur ← nextInBitmap()
7
8 markStep(start):
9   while not isEmpty(worklist)
10      ref ← remove(worklist) /* ref 已被标记过 */
11      for each fld in Pointers(ref)
12          child ← *fld
13          if child ≠ null && not isMarked(child)
14              setMarked(child)
15              if child < start
16                  add(worklist, child)

```

还有一种类似的策略可以应对标记栈溢出的情况，即当标记栈溢出时设置某一标记，同时在后续追踪过程中只设置对象的标记位而不再将其压入标记栈。标记过程会在这种状态下一直运行到标记栈为空，然后我们必须重新找到那些已经标记过但未曾压入标记栈的对象。此时回收器将在堆中查找那些自身已经被标记、但是存在一个或者更多未标记后代的对象，并且对其后代进行标记。最直接的实现方案是在堆中发起一次线性清扫，但清扫位图显然要比检查堆中对象头域中的标记位高效得多。

2.5 懒惰清扫

标记过程的时间复杂度是 $O(L)$ ，其中 L 为堆中存活对象的数量；清扫过程的时间复杂度是 $O(H)$ ，其中 H 为堆空间大小。由于 $H > L$ ，所以我们很容易误认为清扫阶段的开销是整个标记 - 清扫开销的主要部分，但实际情况并非如此。标记阶段指针追踪过程中的内存访问模式是不可预测的，而清扫过程的可预测性则要高得多，同时清扫一个对象的开销也比追

踪的开销小得多。优化清扫阶段高速缓存行为的一种方案是使用对象预取。为避免内存碎片，标记-清扫算法中所用的分配器通常会将大小相同的对象分布在连续空间内（参见 7.4 节），此时回收器可以依照固定步幅对大小相同的对象进行清扫。这一方式不仅支持软件预取，而且可以充分利用现代处理器的硬件预取能力。

接下来我们考虑如何降低甚至消除清扫阶段赋值器的停顿时间。通过观察可以发现，对象及其标记位存在两个特征：第一，一旦某个对象成为垃圾，它将一直都是垃圾，不可能再次被赋值器访问或者复活；第二，赋值器永远不会访问对象的标记位。因此在赋值器工作的同时，清扫器可以并行修改标记位，甚至修改垃圾对象的域，并将其链接到分配结构体中。我们同样也可以使用多个清扫器线程与赋值器并发工作，但更加简单的方案是使用懒惰清扫（lazy sweeping）[Hughes, 1982] 策略。该方案利用分配器来扮演清扫器的角色，即把寻找可用空间的任務转移到 allocate 过程中，从而不再需要单独的清扫阶段。最简单的清扫策略是，allocate 简单地向前移动清扫指针，直到在连续的未标记对象中找到一块足够大的空间，但一次清扫包含多个对象的内存块更具有实用性。

算法 2.5 演示了使用懒惰清扫策略处理整个内存块的方法。分配器通常只会在一个内存块中分配相同大小的对象（在第 7 章中详述），每种空间大小分级（size class）都会对应一个或多个用于分配的内存块，以及一个待回收内存块链表（reclaim list of blocks）。在回收过程中，回收器依然需要将堆中所有存活对象标记，但标记完成后回收器并不急于清扫整个堆，而是简单地将完全为空的内存块归还给块分配器（见算法 2.5 的第 5 行），同时将其其他内存块添加到其所对应空间大小分级的回收队列中。一旦“万物静止”式的回收过程结束，赋值器立即开始工作。对于任意内存分配需求，allocate 方法首先尝试从合适的空间大小分级中分配一个空闲槽（与算法 7.2 所使用的策略相同），如果失败则调用清扫器执行懒惰清扫，即从该空间大小分级的回收队列中取出一个或多个内存块进行清扫，直到满足分配要求为止（见算法 2.5 的第 12 行）。但也可能会出现没有内存块可供清扫，或者被清扫的内存块不包含任何空闲槽的情况，此时分配器便要尝试从更低级别的块分配器中获取新内存块。新内存块通常需要通过设置元数据的方式进行初始化，如构建空闲槽链表或者创建标记字节图，但如果无法获取新内存块，则必须执行垃圾回收。

算法 2.5 块结构堆的懒惰清扫

```

1  atomic collect():
2      markFromRoots()
3      for each block in Blocks
4          if not isMarked(block)                /* 内存块中是否存在已标记对象 */
5              add(blockAllocator, block)        /* 将内存块归还给块分配器 */
6          else
7              add(reclaimList, block)            /* 将内存块添加到懒惰清扫队列 */
8
9  atomic allocate(sz):
10     result ← remove(sz)                        /* 从空间大小 sz 中进行分配 */
11     if result = null                            /* 该空间大小分级中是否没有空闲槽 */
12         lazySweep(sz)                          /* 仅执行少量清扫工作 */
13     result ← remove(sz)
14     return result                             /* 如果仍然为空，则调用 collect */
15
16 lazySweep(sz):
17     repeat
18         block ← nextBlock(reclaimList, sz)
19         if block ≠ null

```

```

20     sweep(start(block), end(block))
21     if spaceFound(block)
22         return
23     until block = null          /* 该空间大小分级的可回收块列表为空 */
24     allocSlow(sz)              /* 获取一个空闲内存块 */
25
26     allocSlow(sz):              /* 分配慢速路径 */
27         block ← allocateBlock() /* 从块分配器中获取内存块 */
28         if block ≠ null
29             initialise(block, sz)

```

在块结构堆（例如从多个空间大小分级中进行分配）中进行懒惰清扫有一个细微的问题需要注意。Hughes[1982] 使用连续堆，并且确保在因内存耗尽而再次引发垃圾回收之前，分配器已经完成所有空闲节点的清扫。但懒惰清扫不能满足这一要求，因为几乎可以肯定的是，在分配器完成对每个空间大小分级的待回收内存块的清扫之前，必然会存在某个空间大小分级（及其所有的空闲内存块）先被耗尽。这将导致两个问题：第一，未清扫内存块中的垃圾对象不会被回收，进而产生内存泄漏，但如果未清扫块中包含存活对象，则泄漏是无害的，因为一旦赋值器从该空间大小分级中分配对象，这些槽便可以得到回收。第二，如果未清扫内存块中的对象后来都成为垃圾，那么我们就失去了将内存块整体回收的机会，从而只能使用开销更大的基于空间大小分级的方法进行清扫。

最简单的解决方案是在标记开始之前完成堆中所有内存块的清扫，但更好的策略是增加内存块得到懒惰回收的几率。Garner 等 [2007] 以容许一定的内存泄漏为代价来避免过早的内存块清扫，他们在 Jikes RVM/MMTk[Blackburn 等, 2004] 中实现了这一方案。该算法使用一个有界整数而非一个位来标记对象，这通常不会增加额外的空间开销，因为如果将标记位置于对象头部，通常会有多于一位的可用空间，而如果使用独立的标记图，通常使用的是字节图而非位图。回收器内部记录一个标记值，并使用该值设置存活对象的标记值。每个回收周期开始时，回收器都会将内部标记值加 1，然后模 2^K （ K 为对象标记值的位数），该值溢出后将绕回到零。回收器可以据此判定某一对象是在本次还是以以往的回收过程中得到标记，只有当对象的标记值与回收器当前标记值相等时，才认为该对象是在本次回收过程中标记的。标记值发生回绕是安全的，因为在该值发生回绕之前，堆中存活对象要么是未标记的（即这些对象创建于上一次回收之后），要么拥有最大的标记值。如果某一对象的标记值与回收器的下一个标记值相等，那么该对象必然是在 2^K 之前的那次回收过程中就已经标记了，因此该对象必然是标记器无法访问到的浮动垃圾。内存块标记（block marking）策略可以在一定程度上解决这种潜在的泄漏问题。MMTk 回收器在标记一个对象的同时也会对其所在的内存块进行标记。如果内存块中没有一个对象的标记值与回收器的当前标记值相同，那么回收器将不会标记该内存块，而是像算法 2.5 的第 5 行那样将其整体回收。在“对象成簇出现、成批死亡”的统计规律下，这一策略将十分有效。

懒惰回收存在多种优点：对象槽通常会在完成清扫后立即得到复用，因而提升了程序的局部性；懒惰清扫策略将标记 - 清扫算法的复杂度降低到与堆中存活对象成正比的水平，这一点与第 4 章将要提到的半区复制式垃圾回收是相同的。Boehm[1995] 特别提到，在标记 - 复制算法能够最优发挥功效的场景下，标记 - 懒惰清扫算法也能达到最佳表现，即如果堆中大部分空间都是空闲的，那么通过懒惰清扫来查找未标记对象将是最快的。在实际应用中，赋值器初始化对象的开销主要取决于清扫和分配过程的开销。

2.6 标记过程中的高速缓存不命中问题

我们已经看到，预取技术可以改进清扫阶段的性能。下面我们将考察如何利用预取技术提升标记阶段的性能。2.4 节提到，使用位图标记可以减少由于读取和设置标记位导致的高速缓存不命中，但在追踪过程中对未标记对象的域的读取也会受到高速缓存不命中问题的影响。此时使用位图标记所带来的潜在缓存好处很容易被加载对象域的开销所掩盖。

如果某个对象不包含指针，则无须加载其任意一个域。尽管不同语言 and 不同应用程序之间的差别较大，但无论如何，堆中都很可能存在相当多的不包含用户自定义指针的对象。对象是否包含指针取决于该对象的类型，一种判定的方式是根据对象头部中的类型信息槽决定，也可以根据对象的地址（address）获取其类型信息，例如当同种类型的对象在堆中集中排列时。Lisp 系统通常使用页簇分配（big bag of pages allocation, BiBoP）技术进行分配，即仅在一页中分配一种类型的对象（例如 cons 单元），从而将类型信息与页而非单个对象相关联 [Foderaro 等, 1985]。同样也可以简单地将全指针对象和无指针对象隔离。历史上还曾经出现过将类型信息编码进指针的方案 [Steenkiste, 1987]。

Boehm[2000] 发现标记过程的开销决定着回收时间：在 Intel Pentium III 系统中，预取对象第一个指针域的开销通常会占到标记该对象总开销的三分之一。为此 Boehm 提出一种灰色预取（prefetching on gray）技术，即当对象为灰色时（即被添加到标记栈时，见算法 2.2 中第 20 行），预取其第一个高速缓存行中的数据，如果被扫描的对象很大，则预取适当数量的高速缓存行。然而，这种技术依赖于预取时间，如果过早地进行高速缓存行预取，则数据很可能在得到使用之前就被换出，而过晚地预取则会导致高速缓存不命中。

Cher 等 [2004 年] 观察到，高速缓存行的预取遵循广度优先（breadth-first）、先进先出（first-in, first-out, FIFO）顺序，而标记-清扫算法对图的遍历却遵循深度优先（depth-first）、后进先出（last-in, first-out, LIFO）顺序。他们的解决方案是在标记栈之前增加一个先进先出队列（如图 2.2、算法 2.6 所示）。与普通的标记过程类似，用 mark 方法将对象加入工作列表的方式依然是将其压入标记栈，而当从工作列表中获取一个对象时，回收器将栈顶对象弹出并将其追加到队尾，而用 mark 方法所处理的对象则是队列中最老的元素。回收器会对从栈顶弹出的对象进行预取，队列的长度将决定预取的距离。在对象从栈中弹出之后，对其进行少量的预取工作可以确保要扫描的对象已经加载到高速缓存中，从而减少高速缓存不命中对标记过程的影响。

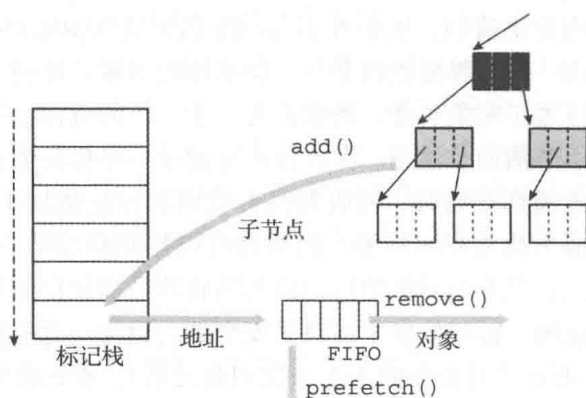


图 2.2 使用先进先出预取缓冲器的标记过程。将对象加入工作列表的方法依然是将其压入标记栈，然而当从工作列表中移出一个对象时，回收器先从预取缓冲区中获取最老的元素，然后再从标记栈顶弹出一个对象并将其补入 FIFO。在对象在预取缓冲区中停留的这段时间内，回收器会将其预取到高速缓存中

算法 2.6 基于先进先出预取缓冲区的标记过程

```
1 add(worklist, item):
```

```

2   markStack ← getStack(worklist)
3   push(markStack, item)
4
5   remove(worklist):
6     markStack ← getStack(worklist)
7     addr ← pop(markStack)
8     prefetch(addr)
9     fifo ← getFifo(worklist)
10    prepend(fifo, addr)
11    return remove(fifo)

```

基于先进先出队列进行对象预取可以确保 mark 方法在扫描对象时免受高速缓存不命中的影响（见算法 2.2 中第 16 ~ 17 行），但回收器检测和设置对象子节点标记位的操作依然可能受到高速缓存不命中的影响（见算法 2.2 中第 18 行）。Garner 等 [2007] 通过对 mark 方法中追踪循环的重组来达到更好的预取效果。在算法 2.2 中，对象图中的每个存活节点都会经历一次压入标记栈的过程，而另一种方案是将对象图中的每条边都遍历和压入标记栈一次，即对于未被标记对象的后代，回收器并不判断其是否已经被标记过，而是无条件地将其加入工作列表（见算法 2.7）。对象图中边的数量通常大于节点数（Garner 等声称，在典型的 Java 应用程序中边的数量会比节点数量多大约 40%），因此将边加入工作列表的方案不但会花费更多的指令，而且需要更大的工作列表。然而，如果在工作列表中增、删这些额外对象的开销足够小，那么提升高速缓存命中率的收益将超过这些额外工作的开销。算法 2.7 将标记操作从内部循环提出，因此 isMarked 和 Pointers 等可能导致高速缓存不命中的方法所操作的将是已被先进先出队列预取的同一对象 obj。Garner 等发现，即使不使用软件预取，追踪边也比追踪节点更能改善性能，他们猜测，循环结构的变化以及先进先出队列的使用可以提升内存访问模式的可预测性，进而允许更加积极的硬件预测行为。

28

算法 2.7 标记对象图的边而非节点

```

1   mark():
2     while not isEmpty(worklist)
3       obj ← remove(worklist)
4       if not isMarked(obj)
5         setMarked(obj)
6         for each fld in Pointers(obj)
7           child ← *fld
8           if child ≠ null
9             add(worklist, child)

```

2.7 需要考虑的问题

作为第一种被提出的垃圾回收算法 [McCarthy, 1960]，标记-清扫回收可谓历史久远，但其对于开发者和用户仍具有一定的吸引力，相关原因如下。

2.7.1 赋值器开销

最简单的标记-清扫回收器不会给赋值器带来任何额外的读写开销，相比之下，引用计数算法（参见第 5 章）则会引入显著的额外开销。对于那些赋值器和回收器之间需要一定同步操作的更加复杂的回收器，标记-清扫算法也可以作为其基本算法之一。分代回收器（参见第 9 章）、并发回收器和增量回收器（参见第 15 章）都要求赋值器在修改指针时通知回收

器，但对于程序的整体执行时间而言，这一开销通常较小。

2.7.2 吞吐量

使用懒惰清扫策略的标记-清扫回收器通常具有较高的吞吐量。标记阶段的开销相对较小，且其主要取决于指针追踪的开销。对于已发现的存活对象，标记-清扫回收器通常只需要简单地设置一个标记位或者标记字节，而半区复制式回收器（参见第4章）和标记-整理回收器（参见第3章）则必须复制或者移动整个对象。与其他追踪式回收器类似，标记-清扫回收器在执行过程中需要挂起所有赋值器线程，回收停顿时间取决于应用程序的运行及其输入。对于大型系统而言，回收停顿时间极有可能达到数秒甚至更长。

2.7.3 空间利用率

与基于半区复制的回收策略相比，标记-清扫回收显然具有较高的空间利用率。同时与引用计数算法相比，标记-清扫回收的空间利用率仍然较高。将标记位放在对象头部基本不会产生额外的空间开销，而如果使用位图来保存标记位，则额外空间开销的大小取决于对象的字节对齐要求，但其总大小不会超过堆的字节对齐要求的倒数（即堆空间的 $\frac{1}{32}$ 或 $\frac{1}{64}$ ，具体的值取决于堆的组织架构）。相比之下，引用计数算法则需要占用对象头部一个完整的槽来记录引用计数（尽管可以通过限制引用计数最大值的方法来减少所需的空间），复制式回收器必须将可用堆划分为两个大小相等的半区，且在任意时间里只有一个半区可用，因而内存空间利用率更低。但在另一方面，非整理式回收器（如标记-清扫回收以及引用计数）通常需要依赖更加复杂的分配器（如分区适应空闲链表分配），这通常会给回收器带来不可忽视的开销。同时，非整理式回收器通常都会存在内存碎片，这也会对空间利用率造成一定影响。

和其他的追踪式回收算法一样，标记-清扫算法在回收所有死亡对象的空间之前必须先确定空间中所有的存活对象。这是一项昂贵的操作，并且应当偶尔执行。这也意味着追踪式回收器必须在堆中保留一定的空间来执行这项操作。如果存活对象在堆中的比例过大，且内存分配速度较快，那么标记-清扫回收器将会频繁执行回收，从而引发程序性能上的颠簸。对于小到大型的堆，可能需要在堆中保留20%~50%的空间 [Jones, 1996]，而 Herts 和 Berger[2005] 发现，使用标记-清扫回收的 Java 程序如果要达到与显式释放相同的吞吐率，需要比后者多使用数倍的堆空间。

2.7.4 移动，还是不移动

非移动式回收算法的优点和缺点并存。其优点在于，不移动对象的特征使得标记-清扫算法可以用于那些编译器与回收器“不合作”的场景（见第11章）。在缺乏赋值器根集合以及对象域详细类型信息的条件下，回收器不敢贸然将对象移动到新的位置，因为程序的“根集合”可能并不是指针，而是其他类型的用户数据。某些场景下可以使用混合型主体复制回收（hybrid mostly-copying collection）[Bartlett, 1998a ; Hosking, 2006]，此时回收器仍需保守地处理程序根集合（即如果某个槽看起来像是指针，那么就必须假设它确实是指针），并且确保不移动根集合所包含的引用，但回收器却可以获取堆中对象的精确类型信息，因而只要对象没有被钉住（pinned），回收器就可以将其移动。

出于安全考虑，在“不合作”系统中工作的保守式回收器不能修改用户数据，包括对象

头域。这同时也要求回收器尽量将其元数据与用户数据或者其他运行时系统数据隔离，以避免被回收器破坏。基于上述两个原因，使用位图标记的方案要比将标记位保存在对象头部的方案更加安全。

非移动式回收算法的主要问题在于，在长期运行的应用程序中堆可能会逐渐碎片化。非移动式内存分配器所需的空间可能会比所有对象总大小还要大 $O\left(\lg \frac{\max}{\min}\right)$ (\min 和 \max 是对象大小可能达到的最小值和最大值) 倍，因此非整理式回收器会比整理式回收器的回收频率更高。需要注意的是，所有的追踪式回收器都需要在堆中保留足够的额外空间 (20% ~ 50%)，目的是避免回收器出现性能颠簸。

为避免堆空间过度碎片化导致的性能下降，许多使用标记-清扫策略的回收器会定期使用标记-整理等算法进行堆的整理。对于对象大小比例经常变化，或者要分配很多大对象的应用程序而言，这一方法通常更加有效。如果程序在未来的运行过程中会分配更大的对象，可能导致堆中许多小的空洞无法被分配出去，而如果后续分配的对象较小，则这些对象将被分配在较大对象曾经占用的空间中，由此造成的间隙则会被浪费。然而，如果充分利用“对象成簇出现、成批死亡”这一特征来指导堆的管理，可以减轻堆的碎片化趋势 [Dimpsey 等, 2000; Blackburn and McKinley, 2008]。分区适应分配也可以减少对整理过程的依赖。

标记 - 整理回收

内存碎片化[⊖]是非移动式回收器无法解决的问题之一，即：尽管堆中仍有可用空间，但是内存管理器却无法找到一块连续内存块来满足较大对象的分配需求，或者需要花费较长时间才能找到合适的空闲内存。上一章我们提到，对于不需要分配很大对象，或者对象大小相差不大的程序，分配器可以仅在单个内存块中分配相同大小的对象，从而缓解内存碎片问题 [Boehm and Weiser, 1988]。但对于许多长期运行的程序，如果通过非移动式回收器进行内存管理，通常会出现碎片化问题，进而导致程序性能的下降。

在接下来的两章中，我们讨论两种对堆中存活对象进行整理 (compact) 以降低外部碎片 (external fragmentation) 的回收策略。堆整理的最大优势在于，它允许极为快速的顺序分配，即简单地堆上限判断，然后根据所需空间的大小阶跃式地移动空闲指针（我们将在第7章中讨论内存分配机制）。本章我们讨论的是原地整理[⊖]策略，即将对象整理到相同区域的一端。下一章我们将讨论复制式回收策略，即将存活对象从一个区域移动到另一个区域（例如在两个半区之间）。

标记 - 整理算法的执行需要经过数个阶段：首先是标记阶段，其相关内容我们在上一章已经讨论过；然后是整理阶段，即移动存活对象，同时更新存活对象中所有指向被移动对象的指针。在不同算法中，堆的遍历次数、整理过程所遵循的顺序、对象的迁移方式都有所不同。整理顺序 (compaction order) 会影响到程序的局部性。移动式回收器重排堆中对象时所遵循的顺序包括以下3种：

- **任意顺序**：对象的移动方式与它们的原始排列顺序和引用关系无关。
- **线性顺序**：将具有关联关系的对象排列在一起，如具有引用关系的对象，或者同一数据结构中的相邻对象。
- **滑动顺序**：将对象滑动到堆的一端，“挤出”垃圾，从而保持对象在堆中原有的分配顺序。

我们所了解的整理式回收器大多遵循任意顺序或者滑动顺序。任意顺序整理实现简单，且执行速度快，特别是对于所有对象均大小相等的情况。但任意顺序整理很可能会将原本相邻的对象分散到不同的高速缓存行或者虚拟内存页中，从而降低赋值器空间局部性。所有现代标记 - 整理回收器均使用滑动整理顺序，它不改变对象的相对排列顺序，因此不会影响赋值器局部性。复制式回收器甚至可以通过改变对象排布顺序的方式将对象与其父节点或者兄弟节点排列得更近，从而提升赋值器的局部性。近期的一些实验表明，由任意顺序整理导致的对象重排列会大幅降低应用程序的吞吐量 [Abuaiadh 等, 2004]。

整理算法可能会存在多种限制：任意顺序算法只能处理单一大小的对象，或者只能对不同大小的对象分别进行整理；整理过程需要两次甚至三次整堆遍历；对象头部可能需要一个额外的槽来保存迁移信息，这对于通用内存管理器来说是一个显著的额外开销。整理算法可

⊖ 我们将在 7.3 节更为详细地讨论内存碎片问题。

⊖ 一些较旧的论文中经常将其称为 compactifying。

能对指针有特定限制，如指针的引用方向是什么？是否允许使用内部指针？我们将在第 11 章讨论这些问题。

本章我们将研究几种不同类型的整理算法。首先要介绍的是 Edwards 的双指针（two-finger）回收算法 [Saunders, 1974]，尽管该算法实现简单且执行速度快，但它打乱了堆中对象的原有布局。第二种整理式回收算法是一种广泛使用的滑动回收算法，即 Lisp 2 算法。与双指针算法不同，该算法需要在每个对象头部增加一个额外的槽来保存转发地址（forwarding address），即对象移动的目标地址。第三种整理算法是 Jonkers 的引线整理（threaded compaction）算法 [1979]，该算法可以在不引入额外空间开销的情况下实现对象的滑动整理，但它需要两次堆遍历过程，且每次遍历的开销都很高。本章所介绍的最后一种整理算法是单次遍历算法（one-pass algorithm），它是一种现代的滑动回收算法，其执行速度快，且不需要在每个对象上引入额外的空间开销。该算法中的转发地址可以实时计算得出。所有整理式回收算法的执行都遵从如下范式：

```
atomic collect():
    markFromRoots()
    compact()
```

3.1 双指针整理算法

Edwards 的双指针算法 [Saunders, 1974] 属于任意顺序整理算法，其需要两次堆遍历过程，最佳适用场景为只包含固定大小对象的区域。该算法的原理十分简单，即对于某一区域中的待整理存活对象，回收器可以事先计算出该区域整理完成后存活对象的“高水位标记”（high-water mark），地址大于该阈值的存活对象都将被移动到该阈值以下。在算法 3.1 的初始阶段，指针 free 指向区域始端，指针 scan 指向区域末端。在第一次遍历过程中，回收器不断向前移动指针 free，直到在堆中发现空隙（即未标记对象）为止；类似地，不断向后移动指针 scan 直到发现存活对象为止。如果指针 free 和指针 scan 发生交错，则该阶段结束，否则便将指针 scan 所指向的对象移动到指针 free 的位置，同时将原有对象中的某个域（指针 scan 所指向的）修改为转发地址，然后继续进行处理。图 3.1 描述了这一过程，其中对象 A 被移动到新的位置 A'，且在对象 A 中的某个槽（即第一个槽）中记录了 A' 的地址。值得注意的是，该算法的整理质量取决于指针 free 所指向的空隙与指针 scan 所指向的存活对象大小的匹配程度。除非对象大小固定，否则碎片的整理程度一定很低。该阶段完成后，指针 free 将位于存活对象边界。回收器的第二次遍历过程会将指向存活对象边界之外的指针更新为其目标对象中所记录的转发地址，即对象的新位置。

[32]

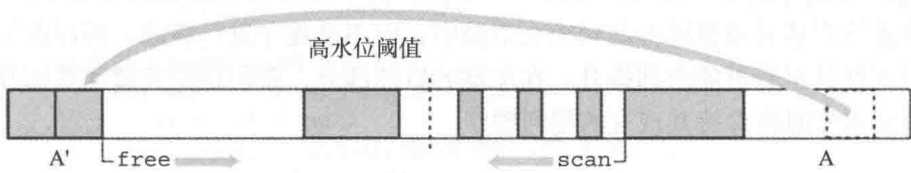


图 3.1 Edwards 的双指针算法。该算法将堆顶的存活对象移动到堆底的空隙中。此处将对象 A 移动到 A'。当指针 free 和指针 scan 交汇时，算法结束

算法 3.1 双指针整理算法

```
1 compact():
```

```

2   relocate(HeapStart, HeapEnd)
3   updateReferences(HeapStart, free)
4
5   relocate(start, end):
6       free ← start
7       scan ← end
8
9       while free < scan
10          while isMarked(free)
11              unsetMarked(free)
12              free ← free + size(free)           /* 寻找下一个空洞 */
13
14          while not isMarked(scan) && scan > free
15              scan ← scan - size(scan)          /* 寻找前一个存活对象 */
16
17          if scan > free
18              unsetMarked(scan)
19              move(scan, free)
20              *scan ← free                       /* 记录转发地址（破坏性地）*/
21              free ← free + size(free)
22              scan ← scan - size(scan)
23
24   updateReferences(start, end):
25       for each fld in Roots                       /* 更新指向被移动对象的根 */
26           ref ← *fld
27           if ref ≥ end
28               *fld ← *ref                       /* 使用上一次遍历过程中记录的转发地址 */
29
30   scan ← start
31   while scan < end                               /* 更新存活区的域 */
32       for each fld in Pointers(scan)
33           ref ← *fld
34           if ref ≥ end
35               *fld ← *ref                       /* 使用上一次遍历过程中记录的转发地址 */
36   scan ← scan + size(scan)                       /* 下一个对象 */

```

33

双指针算法的优势在于简单快速，且每次遍历过程的操作较少。转发地址是在对象移动之后才写入的，所以不会存在任何信息的丢失，因此算法无需使用额外的空间来记录转发地址。该算法支持内部指针。其内存访问模式是可预测的，因此也支持预取（不论是硬件预取还是软件预取），进而可以提升回收器的高速缓存友好性。但是，`relocate` 中指针 `scan` 的移动要求回收器能够“倒退式”地进行堆解析，这便需要将标记位保存在一个独立的位图中，或者在对象分配时即在位图中记录其首地址。不幸的是，双指针算法重排列堆中对象的顺序是任意式的，因此会破坏赋值器的局部性。然而，由于相关对象总是成簇诞生、成批死亡，我们可以将连续存活对象整体移动到较大空隙中，而不是逐个进行移动，所以在某些情况下赋值器的局部性甚至有可能得到提升。在本章的后续部分，我们将研究滑动式回收器，它们在重排列存活对象时将保持其现有的排列顺序。

3.2 Lisp 2 算法

Lisp 2 回收算法（见算法 3.2）是一种历史悠久的回收算法，无论是其原始形态，还是为适应并行回收的改进版本 [Flood 等, 2001]，都得到了广泛应用。该回收器可以用于管理包含多种大小对象的空间，尽管该算法需要三次堆遍历，但是每次遍历要做的工作都不多（只是相对而言，如与引线整理器相比）。虽然所有标记-整理回收器的吞吐量都较低，但是

Cohen 和 Nicolau[1983] 通过一系列复杂研究发现, Lisp 2 算法在他们所研究过的整理式算法中算是最快的。但是他们并没有考虑高速缓存或者页相关行为, 而这一因素通常又十分重要。Lisp 2 算法的主要缺陷在于, 它需要在每个对象头部额外增加一个完整的头域来记录转发地址 (标记位也可以复用该域)。

在标记阶段结束之后的第一次堆遍历过程中, 回收器将会计算出每个存活对象的最终地址 (即转发地址), 并且将其保存在对象的 forwardingAddress 域中 (见算法 3.2)。computeLocations 方法需要 3 个参数: 堆中待整理区域的起始地址、结束地址、整理目标区域起始地址。目标区域通常与待整理区域相同, 但并行回收器可能会为每个线程设定不同的来源和目标区域。用 computeLocations 方法在堆中移动两个指针: 指针 scan 对来源区域中的所有 (存活的或死亡的) 对象进行迭代, 指针 free 指向目标区域中的下一个空闲位置。如果指针 scan 遍历到的对象是存活的, 意味着该对象 (最终) 会被移动到指针 free 所指向的位置。此时回收器将指针 free 写入对象的 forwardingAddress 域, 然后根据对象的大小向前移动指针 free (需要考虑对齐填充)。如果遍历到死亡对象, 则将其忽略。

在第二次堆遍历过程 (算法 3.2 中的 updateReferences 方法) 中, 回收器将使用对象头域中记录的转发地址来更新赋值器线程根以及被标记对象中的引用, 该操作将确保它们指向对象的新位置。在第三次遍历过程中, relocate 最终将每个存活对象移动到其新的目标位置。

算法 3.2 Lisp 2 整理算法

```

1 compact():
2   computeLocations(HeapStart, HeapEnd, HeapStart)
3   updateReferences(HeapStart, HeapEnd)
4   relocate(HeapStart, HeapEnd)
5
6 computeLocations(start, end, toRegion):
7   scan ← start
8   free ← toRegion
9   while scan < end
10    if isMarked(scan)
11      forwardingAddress(scan) ← free
12      free ← free + size(scan)
13      scan ← scan + size(scan)
14
15 updateReferences(start, end):
16   for each fld in Roots /* 更新根 */
17     ref ← *fld
18     if ref ≠ null
19       *fld ← forwardingAddress(ref)
20
21   scan ← start /* 更新域 */
22   while scan < end
23     if isMarked(scan)
24       for each fld in Pointers(scan)
25         if *fld ≠ null
26           *fld ← forwardingAddress(*fld)
27       scan ← scan + size(scan)
28
29 relocate(start, end):
30   scan ← start
31   while scan < end
32     if isMarked(scan)
33       dest ← forwardingAddress(scan)

```



```

34         move(scan, dest)
35         unsetMarked(dest)
36         scan ← scan + size(scan)

```

需要注意的是，遍历的方向（从低地址到高地址）与对象的移动方向（从高地址到低地址）相反，这便可以保证回收器在第三次遍历过程中复制对象时其目的地址已经腾空。某些并行回收器将堆划分为多个内存块，并且在相邻内存块上使用不同的滑动方向，相对于每个内存块都向同一个方向滑动的算法，该算法可以产生较大的对象“聚集”，进而产生更大的空闲内存间隙 [Flood 等, 2001]，图 14.8 即是一个示例。

Lisp 2 算法可以在多方面进行改进：标记-清扫回收器在清扫阶段的数据预取技术也可以应用在 Lisp 2 算法中；在 `computeLocations` 方法的第 10 行之后，回收器可以将相邻垃圾合并，以提升后续遍历过程的性能。

3.3 引线整理算法

Lisp 2 算法的主要缺陷有两个：第一，算法需要三次完整的堆遍历过程；第二，每个对象需要额外的空间来记录转发地址，这两个缺陷可以说是互为因果。滑动整理式回收是一种具有破坏性的操作，存活对象的新副本会覆盖其他存活对象的原有副本，因此在移动对象、更新引用之前，回收器必须记录其转发地址。由于 Lisp 2 算法会占用对象头部的一个槽来记录转发地址，因此它是非破坏性的。在双指针算法中，转发地址记录在存活对象边界之外的已经移动过的对象上，因此它也是非破坏性的，但是它的任意整理顺序无法令人接受。

Fisher[1974] 通过一种不同的策略解决了指针更新问题，即“引线”（threading），该算法不需要任何额外存储，且支持滑动整理。引线算法要求对象头部存在足够的空间来保存一个地址（如果必要可以覆盖头域的其他数据），这一要求并不苛刻，但回收器所记录的地址必须要能与其他值区分，要满足这一要求可能有些困难。最知名的引线算法当属 Morris[1978, 1979, 1982] 的版本，但是 Jonkers[1979] 的版本限制更少（例如在指针方向上）。引线的目的是通过对对象 N 可以找到所有引用了该对象的对象，实现方法是临时反转指针的方向。图 3.2 演示了如何在引线之后找到之前引用了对象 N 的对象。需要注意的是，经过图 3.2b 中的引线操作之后，对象 N 头部 `info` 域的值被写入到对象 A 的一个指针域中，当回收器通过指针追踪来逆引线（`unthread`）、更新引用时，必须要能分辨出对象 A 的这一域中记录的并非引线指针。

Jonkers 的算法需要两次堆遍历过程，第一次遍历实现堆中前向指针[⊖]的引线，第二次遍

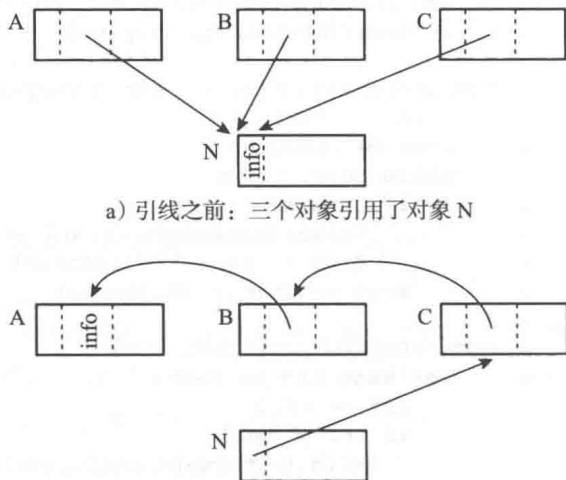


图 3.2 对指针进行引线

⊖ 即从低地址指向高地址的指针。——译者注

历实现堆中后向指针[⊖]的引线（见算法 3.3）。在第一次遍历开始时，回收器先对根进行引线，然后在堆中从头到尾进行扫描，与此同时，将所有存活对象的大小累加，最终以此来更新指针 `free`。在图 3.2 中，如果仅考虑存活对象 N，那么该算法很容易理解：当回收器在第一次遍历过程中遇到对象 A 时，其会对 A 中指向对象 N 的引用进行引线，当遍历到对象 N 时，会完成所有指向对象 N 的前向指针的引线（见图 3.2b）。此时回收器可以沿着对象 N 的这条引线链完成所有指向对象 N 的前向指针的更新，即将它们都改写为指针 `free`，也就是对象 N 未来的新地址。当到达引线链的终点时，回收器将恢复对象 N 头部 `info` 域的值。完成上述步骤之后，还需要增加指针 `free`，并对 N 的所有子节点进行引线。第一次遍历完成之后，所有前向指针都已经指向了对象整理后的新地址，且所有后向指针都已完成引线。第二次遍历过程则会根据后向指针引线链简单地更新指向对象 N 的引用，同时完成对象 N 的移动。

37

算法 3.3 Jonkers 引线整理算法

```

1 compact():
2   updateForwardReferences()
3   updateBackwardReferences()
4
5 thread(ref):                                /* 对引用进行引线 */
6   if *ref ≠ null
7     *ref, **ref ← **ref, ref
8
9 update(ref, addr):                          /* 使用 addr 对所有引用进行逆引线 */
10  tmp ← *ref
11  while isReference(tmp)
12    *tmp, tmp ← addr, *tmp
13  *ref ← tmp
14
15 updateForwardReferences():
16   for each fld in Roots
17     thread(*fld)
18
19   free ← HeapStart
20   scan ← HeapStart
21   while scan ≤ HeapEnd
22     if isMarked(scan)
23       update(scan, free) /* 将所有指向 scan 的前向指针都修改为指针 free */
24       for each fld in Pointers(scan)
25         thread(fld)
26       free ← free + size(scan)
27       scan ← scan + size(scan)
28
29 updateBackwardReferences():
30   free ← HeapStart
31   scan ← HeapStart
32   while scan ≤ HeapEnd
33     if isMarked(scan)
34       update(scan, free) /* 将所有指向 scan 的后向的指针都修改为指针 free */
35       move(scan, free)   /* 将 scan 指向的对象移动到 free */
36       free ← free + size(scan)
37       scan ← scan + size(scan)

```

该算法的主要优点在于不需要额外的空间，尽管其对象头部必须能够容纳一个指针（且该指针必须能与一般的值进行区分），但引线算法也存在不少缺点。该算法需要两次修改对

⊖ 即从高地址指向低地址的指针。——译者注

象的头部，第一次是引线，第二次是逆引线并更新引用。与标记过程类似，Jonkers 的算法中沿着引线链进行遍历的高速缓存友好性较差，而整个算法总共需要三次这样的指针遍历过程（即：标记、引线、逆引线）。Martin[1982] 指出，可以将标记过程与第一次整理过程合并，从而将回收时间减少三分之一，但这也反映了指针追踪以及修改指针域的开销之大。Jonkers 的算法对指针的修改是破坏性的，其本质上是串行的，因此无法用于并发整理。例如在图 3.2b 中，当回收器完成对象 B 中第一个指针域的引线之后，堆中将不再有任何能够反映出该域曾经指向对象 N 这一信息（除非将对象 N 的地址存储在指针链的末端，即在对象 A 的头部中占用一个额外的槽，但这破坏了不使用额外空间的本意）。最后，Jonkers 的算法不支持内部指针（interior pointer），这在某些场景下可能是一个重要问题。Morris[1982] 的引线整理算法虽然支持内部指针，但其代价是要求为每个域分配一个额外的标签位，且第二次整理过程的遍历方向必须与第一次相反（从而引入了堆的可解析性问题）。

3.4 单次遍历算法

如果要将滑动式回收器的堆遍历次数降低到两次（一次标记、一次滑动对象），且避免昂贵的引线开销，那么就必须使用一个额外的表来记录转发地址。Abuaiadh 等 [2004]、Kermany 和 Petrank[2006] 各自独立设计出了可以完全满足这一要求且适用于多处理器的高性能整理算法：前者的算法属于并行式、万物静止式算法（使用多个整理线程）；后者的算法可以配置成并发式回收算法（允许赋值器线程和回收器线程同时执行）和增量式回收算法（定期挂起一个赋值器线程并简单地执行小部分回收工作）。算法的并行、并发、增量部分将在后面的章节中讨论，本节主要关注的是在万物静止条件下的核心整理算法。

这两种算法都需要使用数个额外的表或者向量。与许多回收器类似，标记过程是基于位图（即图 3.3 中的标记向量——译者注）进行的，每个位对应堆中一个内存颗粒（即一个字）。在标记过程中，如果发现存活对象，则设置其所占用空间的第一个和最后一个内存颗粒对应的位。例如在图 3.3 中，回收器会针对存活对象 old 设置标记向量的第 16 位和第 19 位。回收器在后续的整理阶段可以通过对标记向量的分析计算出任意存活对象的大小。

38

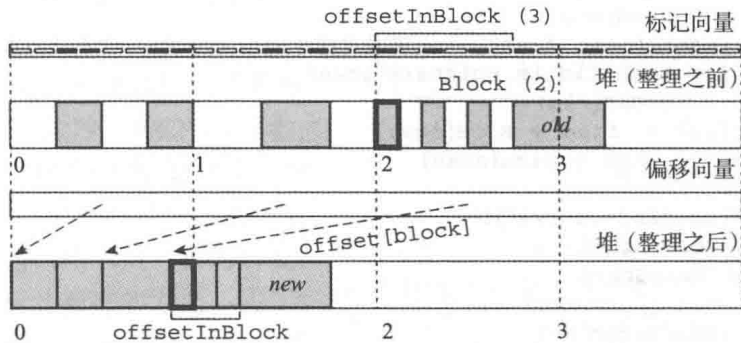


图 3.3 Compressor 回收器所使用的堆（整理前以及整理后）和元数据 [Kermany and Petrank, 2006]。标记位向量（mark-bit vector）中的位反映了每个存活对象的起始和结束地址。偏移向量（offset vector）中的字记录了其对应内存块中第一个存活对象的目标地址。转发地址并未记录，但它可以通过偏移向量和标记位向量计算出来

回收器使用一个额外的表来记录转发地址。如果记录每个对象的转发地址，则会引入难以承受的开销（即使对象已经满足一定的字节对齐要求），因此这两种算法都将堆划分成大

小相等的小内存块（分别是 256 字节和 512 字节）。偏移向量（offset vector）记录了每个内存块中第一个存活对象的转发地址，其他存活对象的转发地址可以通过偏移向量和标记位向量实时计算得出。对于任意给定对象，我们可以先计算出其所在内存块的索引号，然后再根据该内存块在偏移向量和标记位向量中的对应数据计算出该对象的转发地址。因此回收器不再需要两次遍历过程来移动对象和更新指针，转而可以通过对标记位向量的一次遍历来构造偏移向量，然后通过一次堆遍历过程同时完成对象的移动和指针的更新。减少堆的遍历次数可以提升回收器的局部性。下面我们将分析算法 3.4（即 Compressor 算法）的具体实现细节。

算法 3.4 Compressor 回收器的整理算法

```

1 compact():
2   computeLocations(HeapStart, HeapEnd, HeapStart)
3   updateReferencesRelocate(HeapStart, HeapEnd)
4
5 computeLocations(start, end, toRegion):
6   loc ← toRegion
7   block ← getBlockNum(start)
8   for b ← 0 to numBits(start, end)-1
9     if b % BITS_IN_BLOCK = 0                /* 是否跨越了块边界? */
10      offset[block] ← loc                    /* 第一个对象将移动到 loc */
11      block ← block + 1
12      if bitmap[b] = MARKED
13        loc ← loc + BYTES_PER_BIT          /* 根据存活对象的大小增加 */
14
15 newAddress(old):
16   block ← getBlockNum(old)
17   return offset[block] + offsetInBlock(old)
18
19 updateReferencesRelocate(start, end):
20   for each fld in Roots
21     ref ← *fld
22     if ref ≠ null
23       *fld ← newAddress(ref)
24   scan ← start
25   while scan < end
26     scan ← nextMarkedObject(scan)          /* 使用位图 */
27     for each fld in Pointers(scan)         /* 更新引用 */
28       ref ← *fld
29       if ref ≠ null
30         *fld ← newAddress(ref)
31     dest ← newAddress(scan)
32     move(scan, dest)

```

在标记过程结束之后，computeLocations 方法将通过对标记位向量的遍历来计算偏移向量。从本质上讲，这一过程与 Lisp 2（见算法 3.2）中的计算方法一致，但它不需要访问堆中对象。我们以图 3.3 中 block 2 内的第一个存活对象为例（即图中加粗的方块），block 0 中的第 2、3、6、7 位被设置，block 1 中的第 3、5 位被设置（本例中，每个内存块包含 8 个槽），这表示在该对象之前已经有 7 个内存颗粒（字）在位图中得到了标记，因此 block 2 中的第一个存活对象将被移动到堆中第 7 个槽中。回收器将这一地址记录在与该块对应的偏移向量中（图中标有 offset[block] 的虚线）。

完成偏移向量的计算后，回收器将更新根以及存活域，并使其指向对象的新地址。在 Lisp 2 算法中，由于迁移信息记录在堆中，且堆中对象的移动会破坏原有对象的迁移信

息，因此回收器需要将更新引用和移动对象的过程分开。但在 Compressor 算法中，转发地址可以快速通过标记位向量和偏移向量实时计算得到，因而无须将其保存在堆中，于是回收器可以在单次遍历过程中同时完成对象的迁移以及引用的更新，即算法 3.4 中的 `updateReferencesRelocate` 方法。对于堆中任意给定地址的对象，Compressor 回收器均可通过 `newAddress` 方法获取其内存块编号（通过移位和掩码操作），并且将该值作为偏移向量的索引值来获取其中第一个存活对象的转发地址，然后再借助标记向量获取内存块中存活对象的数量及大小，并据此增加偏移。这一操作可以通过查表的方式在常数时间内完成，例如，在图 3.3 中，对象 `old` 在内存块的已标记槽中的偏移量为 3，那么该对象的目标地址将是第 10 个槽，即：`offset[block]=7` 加上 `offsetInBlock(old)=3`。

3.5 需要考虑的问题

3.5.1 整理的必要性

标记-清扫回收会比复制式回收（将在下一章描述）等其他回收算法占用更少的内存。由于标记-清扫算法不会移动对象，所以它只需要确定根集合（的一个超集）而无须对其进行修改。对于内存较小，或者运行时系统无法提供精确类型信息的场景，这些特点都十分重要（见 11.2 节）。

作为一种非移动式回收算法，标记-清扫算法很容易受到内存碎片问题的影响。使用顺序适应分配（见 7.4 节）等节约型分配策略可能会减缓碎片化问题，但这要求程序不分配太多较大对象，且对象之间大小差异不大。但如果在对象大小变化较大，且需要长期运行的程序中使用通用的非移动式回收器，碎片化几乎必然会是一个问题，因此许多生产环境下的 Java 虚拟机都使用可以进行堆整理的移动式回收器。

3.5.2 整理的吞吐量开销

在整理式堆中进行顺序分配的速度很快。如果堆可用内存相对较大，则标记-整理算法是一个合适的移动式回收策略，其内存需求量仅为复制式回收器的一半。与复制式回收算法相比，Compressor 等算法更适合多回收器线程的场景（我们将在 14 章进行介绍）。当然，这些优点是附带有一定代价的。与标记-清扫或者复制式回收器相比，标记-整理回收器的速度通常较慢。许多整理算法需要额外的空间开销，或者对赋值器有一定要求。

与标记-清扫或者复制式回收算法相比，标记-整理算法需要更多次堆遍历过程，因而其吞吐量较差（Compressor 算法是一个特例）。每次遍历过程的开销都很大，因为这不仅需要多次访问类型信息以及对象的指针域，而且正如第 2 章所提到的，“指针追踪”的开销往往更大。一个通用的解决方案是，尽量长久地使用标记-清扫算法进行回收，仅当碎片化达到一定程度时才使用标记-整理回收 [Printezis, 2001; Soman 等, 2004]。

3.5.3 长寿数据

在移动式回收器中，长寿对象甚至是永生对象聚集在堆底的情况并不罕见。复制式回收器在处理这些对象时的表现很差，它只会重复地将其从一个半区复制到另一个半区。分代式回收器（将在第 9 章介绍）可以将这些对象移动到一个很少进行回收的区域，从而较好地实现长寿对象的处理。然而，分代的解决方案可能并不适用于堆空间较小的场景，因为如果要

对分代式回收器中最老的一代进行回收，仍需处理长寿对象。相比之下，标记 - 整理回收则可以简单地选择不去整理这一“沉积区”内的对象。Hanson[1977]在其 SITBOL 系统中首先发现，这些对象往往聚集在该系统中“临时对象区”的底部。他的解决方案是动态跟踪“沉积区”的高度，只要不是绝对必要，就简单地避免对其进行处理，付出的代价只是少量的内存碎片。Sun Microsystems 的 HotSpot Java 虚拟机使用标记 - 整理作为其最老一代的默认回收器，它同样也会避免整理堆内用户配置的“密集前缀”（dense prefix）区域中的对象 [Sun Microsystems, 2006]。如果使用位图标记，那么可以通过位图简单地计算出达到指定密度的存活对象前缀。

3.5.4 局部性

标记 - 整理回收器可能会保留对象在堆中原有的分配顺序，也可能以任意顺序进行重排列。尽管任意顺序回收器会比其他种类的标记 - 整理回收器更快，且无须额外的空间开销，但随意扰乱对象排列顺序很可能会影响赋值器的局部性。在某些系统中，滑动式标记 - 整理回收器存在一些其他的优点：对于在程序某一点之后分配的空间，可以简单地通过移动空闲指针的方式在常数时间内将其释放。

41

3.5.5 标记 - 整理算法的局限性

研究者们提出了多种不同类型的标记 - 整理算法。Jones[1996] 的第 5 章对许多较老的整理策略进行了较为详尽的描述，这些算法大多都存在一些不良的或者不可接受的缺陷。需要考虑的问题包括记录转发指针要付出多大的空间开销（尽管这一开销比复制式回收器要小）。某些整理算法对赋值器具有特定限制：双指针算法等简单整理算法只能处理固定大小的对象，将对象依照大小进行分级当然是可以的，但既已如此又何须进行整理；引线整理要求能够将指针和暂存在指针域中的非指针临时值进行区分，由于引线算法会（临时性地）破坏针域信息，因而不适用于并发回收器；Morris[1978, 1979, 1982] 的引线整理算法对引用可能的指向方向存在限制；最后，除了双指针算法，大多数整理算法都不支持内部指针。

42

复制式回收

标记-清扫回收的开销较低，但其可能受到内存碎片问题的困扰。在一个设计良好的系统中，垃圾回收通常只会占用整体执行时间的一小部分，赋值器的执行开销将决定整个程序的性能，因此应当设法降低赋值器的开销，特别是应当尽量提升它的分配速度。标记-整理回收器可以根除碎片问题，而且支持极为快速的“阶跃指针”（bump a pointer）分配（见第 7 章），但它需要多次堆遍历过程，进而显著增加了回收时间。本章将介绍第三种追踪式回收算法：半区复制（semispace copying）[Fenichel and Yochelson, 1969; Cheney, 1970]。回收器在复制过程中会进行堆整理，从而可以提升赋值器的分配速度，且回收过程只需对存活对象遍历一次。其最大的缺点在于，堆的可用空间降低了一半。

4.1 半区复制回收

基本的复制式回收器会将堆划分为两个大小相等的半区（semispace），分别是来源空间（fromspace）和目标空间（tospace）。为了简单起见，算法 4.1 假定堆是一块连续的内存空间，但这并非强制性要求。当堆空间足够时，在目标空间中分配新对象的方法是根据对象的大小简单地增加空闲指针[⊖]，如果可用空间不足，则进行垃圾回收。回收器在将存活对象从来源空间复制到目标空间之前必须先将两个半区的角色互换（见算法 4.2 中的第 2 行）。在回收过程中，回收器简单地将存活对象从来源空间中迁出；在回收完成后，所有存活对象将紧密排布在目标空间的一端。在下一轮回收之前，回收器将简单地丢弃来源空间（以及其中的对象），但在实际应用中基于安全考虑，许多回收器在初始化下一轮回收过程之前都会先将该区域清零（见第 11 章中讨论运行时系统接口的内容）。

算法 4.1 半区复制回收：初始化以及分配（为了简单起见，假定堆是单个连续区域）

```

1 createSemispaces():
2     tospace ← HeapStart
3     extent ← (HeapEnd - HeapStart) / 2           /* 半区大小 */
4     top ← fromspace ← HeapStart + extent
5     free ← tospace
6
7 atomic allocate(size):
8     result ← free
9     newfree ← result + size
10    if newfree > top
11        return null                               /* 信号：“内存耗尽” */
12    free ← newfree
13    return result

```

在回收过程的初始化完成之后，半区复制回收器首先将根对象复制到目标空间，并以此来填充工作列表（算法 4.2 中的第 4 行）。已复制但尚未得到扫描的对象为灰色，其每个指针

⊖ 注意：我们所说的分配和复制过程都忽略了字节对齐和字节填充要求，也忽略了对象复制前后可能存在不同格式的情况，例如，每个 Java 对象都有一个明确的哈希值。

域要么为空，要么引用了某个位于来源空间的对象。回收器扫描灰色对象的每个指针域，并将其更新到目标对象在目标空间中的新副本。当遍历到来源空间中的某一对象时，copy 方法首先检查该对象是否已完成迁移（即是否已存在转发地址），如果没有，则将该对象复制到目标空间中指针 free 所指向的地址，同时根据对象的大小增加指针 free（与分配过程类似）。对于存活对象在目标空间中的对应副本，回收器必须能够保持其原有的拓扑关系，因此当回收器将对象复制到目标空间时，会将其转发地址记录在来源空间内的原有对象中（算法 4.2 中的第 34 行）。forward 方法在对目标空间中的域进行扫描时会使用目标对象的转发地址来更新该域，如果目标对象的转发地址尚不存在，则对该对象进行复制（算法 4.2 中的第 22 行）。当回收器完成对目标空间中所有对象的扫描时，回收过程结束。

与标记 - 整理回收不同，半区复制回收无须在对象头部中引入额外空间。由于来源空间中的对象在复制完成后便不再使用，所以其每个槽都可以用于记录转发地址（至少在万物静式回收中如此）。因此复制式回收甚至适用于不包含头部的对象。

算法 4.2 半区复制式回收

```

1  atomic collect():
2      flip()
3      initialise(worklist)                /* 将工作列表初始化为空 */
4      for each fld in Roots                /* 复制根 */
5          process(fld)
6      while not isEmpty(worklist)          /* 复制传递闭包 */
7          ref ← remove(worklist)
8          scan(ref)
9
10 flip():                                  /* 翻转半区 */
11     fromspace, tospace ← tospace, fromspace
12     top ← tospace + extent
13     free ← tospace
14
15 scan(ref):
16     for each fld in Pointers(ref)
17         process(fld)
18
19 process(fld):                            /* 使用目标空间中新副本的地址来更新域 */
20     fromRef ← *fld
21     if fromRef ≠ null
22         *fld ← forward(fromRef) /* 使用目标空间中新副本的地址进行更新 */
23
24 forward(fromRef):
25     toRef ← forwardingAddress(fromRef)
26     if toRef = null                    /* 尚未得到复制（尚未标记） */
27         toRef ← copy(fromRef)
28     return toRef
29
30 copy(fromRef):                            /* 复制对象，返回转发地址 */
31     toRef ← free
32     free ← free + size(fromRef)
33     move(fromRef, toRef)
34     forwardingAddress(fromRef) ← toRef    /* 标记 */
35     add(worklist, toRef)
36     return toRef

```

4.1.1 工作列表的实现

与其他追踪式回收器类似，半区复制需要一个工作列表来记录待处理对象。工作列表有

多种实现方式，每种方式的对象图遍历顺序以及空间需求各不相同。Fenichel 和 Yochelson [1969] 的策略是将工作列表作为一个简单的辅助栈，十分类似于第 2 章所描述的标记-清扫回收器所使用的标记栈，当栈为空时，复制过程结束。

Cheney[1970] 提出了一种十分优雅的算法：Cheney 扫描 (Cheney scanning)，该算法利用目标空间中的灰色对象实现先进先出队列。该算法仅需要一个指针 `scan` 来指向下一个待扫描对象，除此之外不再需要任何额外空间。在半区翻转完成后，指针 `free` 和指针 `scan` 均指向目标空间的起始地址（见算法 4.3 中的 `initialise` 方法）。完成根对象的复制后，指针 `scan` 和指针 `free` 之间的灰色对象（已完成复制但未完成扫描）便构成了工作列表。随着目标空间中对象域的扫描以及更新，指针 `scan` 不断向前迭代（见算法 4.3 中的第 9 行）。当工作列表为空，也就是指针 `scan` 与指针 `free` 重合时，回收完成。该算法的实现非常简单，要确定回收过程是否完成，仅需要通过 `isEmpty` 方法判断指针 `scan` 和指针 `free` 是否重合，`remove` 方法只是简单地返回指针 `scan`，`add` 方法则无须执行任何操作。

算法 4.3 用 Cheney 工作列表进行复制

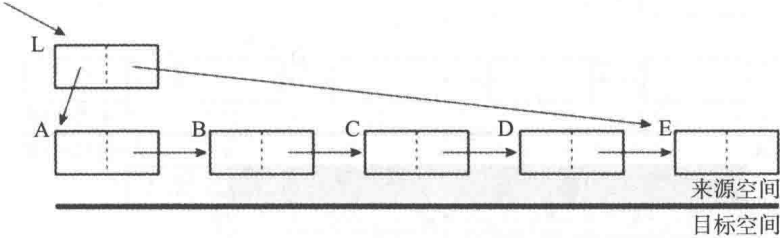
```

1  initialise(worklist):
2      scan ← free
3
4  isEmpty(worklist):
5      return scan = free
6
7  remove(worklist):
8      ref ← scan
9      scan ← scan + size(scan)
10     return ref
11
12 add(worklist, ref):
13     /* 空 */

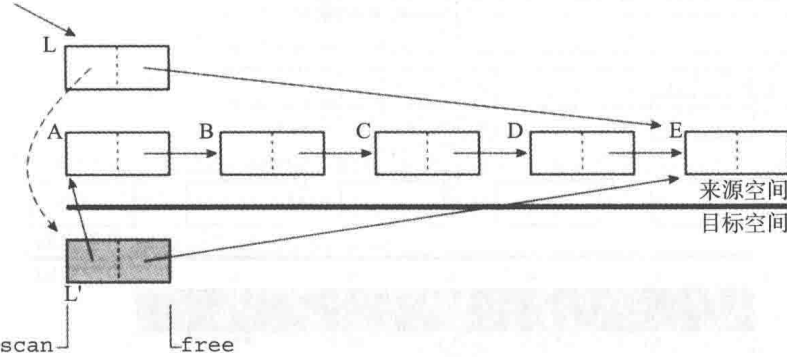
```

4.1.2 示例

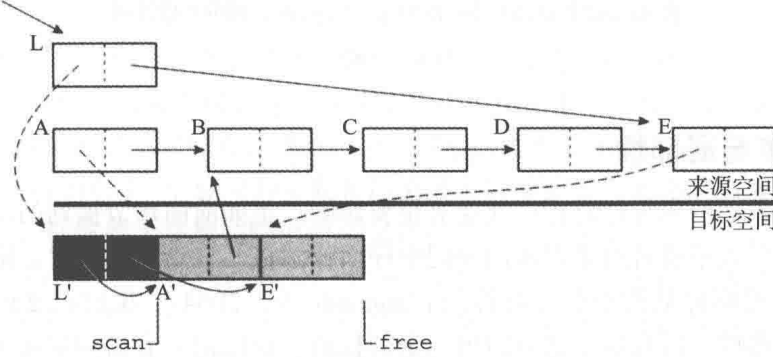
图 4.1 展示了使用 Cheney 扫描复制对象 `L` 的一个示例，该对象是链表结构的头节点，它包含指向表头和表尾的指针。图 4.1a 展示了在回收开始之前来源空间的状态。在回收开始时，回收器翻转半区，将根直接可达的对象 `L` 复制到目标空间（增加指针 `free`），同时将对象 `L` 的新地址（即 `L'`）写入 `L`（例如覆盖其第一个域），然后将指针 `scan` 指向目标空间中的第一个对象（见图 4.1b），此时回收器已经做好了复制根的传递闭包（transitive closure）的准备。`scan` 指针指向该过程的第一个对象。`L'` 持有来源空间中对象 `A` 和对象 `E` 的引用，回收器将它们复制到目标空间中指针 `free` 所指向的地址（并增加指针 `free`），同时也将 `L'` 中的引用更新以指向对象 `A'` 和 `E'`（见图 4.1c），然后将指针 `scan` 移动到下一个灰色对象。需要注意的是，在回收器完成对象 `L'` 的处理后，`L'` 在概念上将成为黑色，而将要扫描的对象 `A'` 和 `E'` 则变成灰色。在目标空间中的对象身上重复这一过程，直到指针 `scan` 和指针 `free` 汇合为止（见图 4.1f）。注意，在图 4.1e 中，`D'` 引用了已完成复制的对象 `E`，回收器使用对象 `E` 中记录的转发地址来更新 `D'` 中的引用，从而保持了对象图的拓扑形状。与其他追踪式算法一样，复制式垃圾回收可以处理包括环状数据结构在内的任何形状的图，也可以正确地保持共享关系。



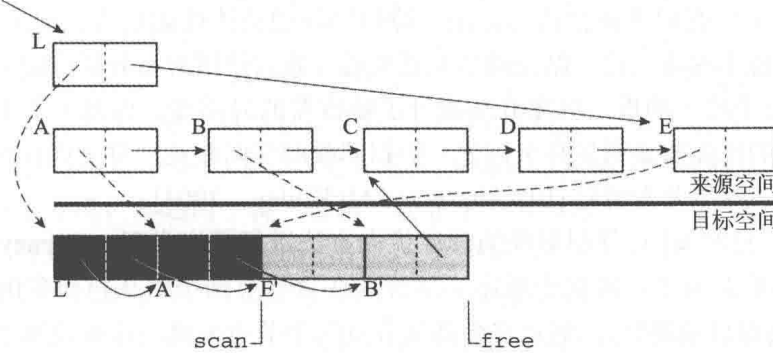
a) 在回收之前来源空间的状态



b) 复制根对象 L

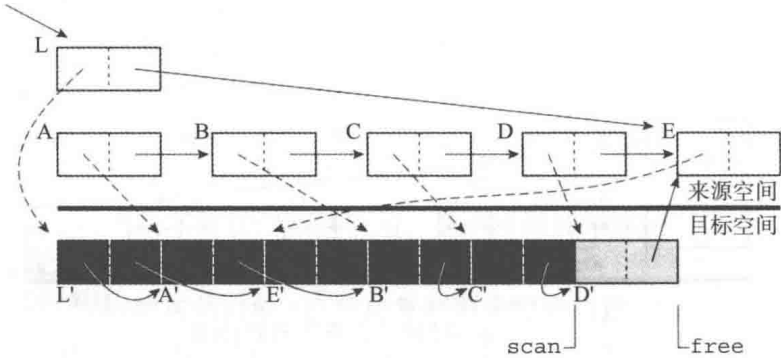


c) 扫描对象 L 的副本

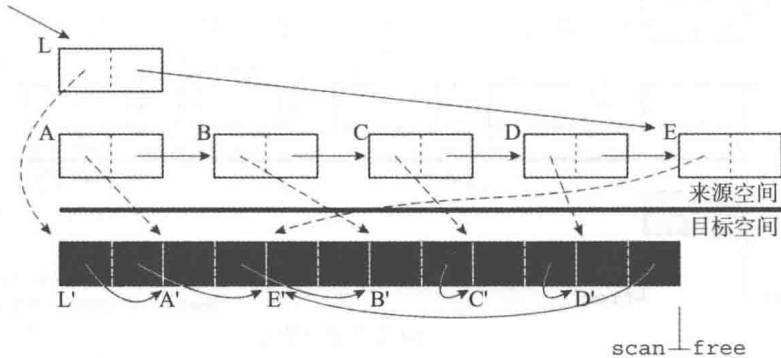


d) 扫描对象 A 以及其他对象的副本

图 4.1 复制式垃圾回收：示例



e) 扫描对象 C 的副本



f) 扫描对象 D 的副本。此时 scan=free，说明回收结束

图 4.1 (续)

4.2 遍历顺序与局部性

赋值器和回收器的局部性对程序性能有重要影响。正如前面章节提到的，如果回收器忽略对象之间的指针关系或者对象原本的分配顺序而任意将其移动到新位置，则很可能降低赋值器的局部性，进而降低整个程序的性能 [Abuaiadh 等，2004]。我们需要在赋值器的局部性、回收器的局部性、回收频率之间做出一定的权衡。以标记-清扫和复制式回收的对比为例：在同等条件下，标记-清扫回收的可用堆大小是复制式回收的两倍，因此其回收次数会比后者少一半，于是我们可能会认为标记-清扫回收的整体性能更优。Blackburn 等 [2004] 发现，对于空间较小的堆，这一结论确实是正确的（他们使用的是分区适应分配以及非移动式回收器），但对于较大的堆，顺序分配提升了赋值器的局部性，提升了各个层次的缓存命中率，其所带来的性能收益明显高于标记-清扫回收的空间收益。对于更新频率较高的新分配对象而言，这一效果尤为明显 [Blackburn and McKinley, 2003]。

Blackburn 等 [2004a] 对复制对象的深度优先方法进行研究发现，Cheney 的复制式回收器遍历顺序在本质上属于广度优先顺序，尽管其在目标空间中对灰色对象的扫描是线性的（即其访问模式具有可预测性），但是它会将父节点与子节点分离，从而破坏了赋值器的局部性。对于图 4.2a 所示的对象布局，图 4.2b 中的表对比了对同一对象布局使用不同顺序进行遍历的结果，每一行展示了不同遍历顺序下对象在目标空间中的最终排列形式。对第 2 行进行观察可知，在广度优先顺序下，只有对象 2 和 3 距其父节点较近。本节将深入探讨遍历顺序及其对赋值器局部性的影响。

46
48

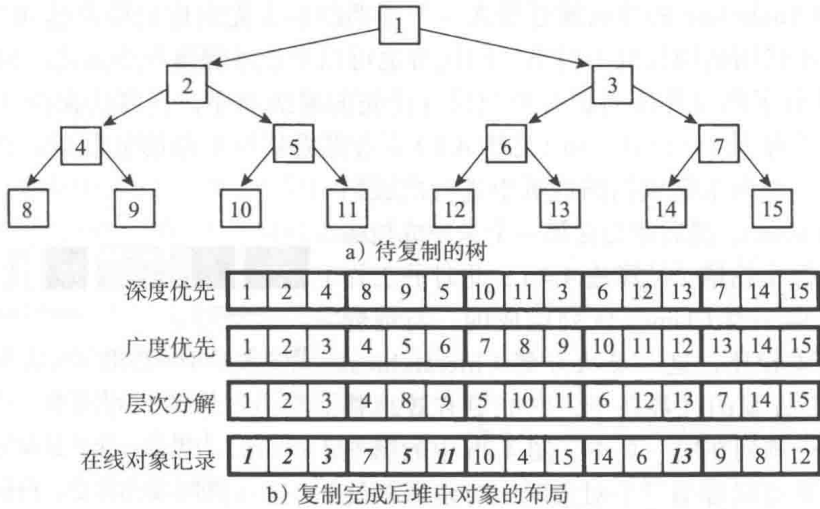


图 4.2 使用不同的遍历顺序来复制一棵树。假设每个内存页（图中用粗线框标出）可以容纳三个对象，每一行展示了不同遍历顺序下对象在目标空间中的布局方式。对于在线对象记录（online object recordering）策略，质数域（粗斜体）被认为是热点

White[1980] 在很早以前就建议利用垃圾回收器来提升赋值器的性能。复制式回收器和整理式回收器都会移动对象，因此可以潜在地影响赋值器的局部性。对于标记 - 整理算法而言，滑动顺序通常是最优的，因为它保持了赋值器分配对象时建立的顺序。滑动顺序是一种安全的、保守的策略，但是否可以做得更好呢？标记 - 整理算法进行堆紧缩的方式，无非是将对象移动到空洞中（任意顺序整理），或者滑动存活对象（仅覆盖垃圾或者覆盖已经移动过的对象），因此它无法通过对象重排列来进一步提升赋值器的局部性。对于将存活对象迁移到新空间且不破坏原有数据的复制式回收，其可以通过对象的重排列来提升赋值器的局部性。

但是，我们无法找到一个最优的对象布局来最大限度地提升程序的高速缓存命中率，其原因有二：首先，回收器无法预知赋值器未来将会以何种方式访问存活对象；其次，Petrunk 和 Rawitz[2002] 指出，对象排列问题是一个 NP 完全问题，也就是说，即使可以完全预知赋值器未来访问对象的次序，也无法找到一个高效算法来计算出最优的排列方式。唯一的办法是使用启发式方法。通过程序过往的行为来预测其未来的行为是一种可行方案。一些研究者假定程序在不同输入下行为都是相似的，进而采取在线分析（profiling）策略 [Calder 等，1998]，也有研究者假定程序在连续两个时间区间内的行为不会发生变化，进而使用在线采样（online sampling）策略 [Chilimbi 等，1999]。另一种启发式方法是保持对象在分配时的顺序，就像滑动整理那样。第三种方法是尝试将子节点靠近它的某个父节点排列，因为访问子节点的唯一途径是经过该节点的一个父节点。Cheney 算法使用广度优先遍历，从而导致具有相关性的对象分离，即趋向于将“远亲”而非父子节点排列在一起，而深度优先遍历则趋向于将子节点与其父节点排列得更近（图 4.2b 中的第一行）。

对于不同复制顺序对赋值器局部性的影响，早期的研究主要集中在减少缺页异常（page fault）方面，其目的是将相关对象排列在同一内存页中。Stamos 在对 Smalltalk 系统进行模拟时发现，在换页行为方面，深度优先顺序虽然比广度优先顺序有一定的改进，但却比滑动顺序要差 [Stamos, 1982 ; Blau, 1983 ; Stamos, 1984]。Wilson 等 [1991] 则认为，这些模拟都忽略了 Lisp 和 Smalltalk 程序真正的拓扑结构，即程序倾向于创建宽而浅的树，树的根节点通常位于哈希表中，且为避免冲突，对象的键通常会被打散。

Fenchel 和 Yochelson 的算法通过引入一个辅助的后进先出标记栈来达到深度优先遍历顺序, 但即使不使用辅助栈且不付出空间代价也可以实现准深度优先遍历。Moon[1984] 对 Cheney 算法进行了修改并使其拥有近似深度优先的遍历顺序, 新算法在指针 `scan` 的基础上引入了第二个指针 `partialScan` (见图 4.3)。当完成某一对象的复制后, 该算法先在目标空间内最后一个尚未完成扫描的页中进行次级扫描 (secondary scan), 然后才会在第一个未完成扫描的页中继续进行主扫描 (见算法 4.4)。此时的工作列表实际上是由一对 Cheney 队列组成的。与纯粹的广度优先搜索相比, 这一层次分解 (hierarchical decomposition) 方案的优势在于, 它能更有效地将父节点与子节点排列在同一页中。图 4.2b 中的第三行展示了在一页可以容纳三个对象时, 对整棵树使用层次分解算法进行复制之后的状态。

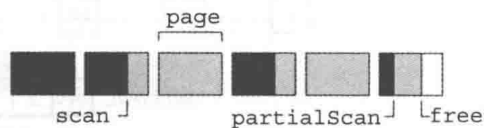


图 4.3 Moon 的准深度优先复制。每一块代表一个内存页。已经扫描过的域为黑色, 完成复制但尚未得到扫描的对象为灰色, 白色为空闲空间

算法 4.4 准深度优先复制 [Moon, 1984] (假设对象不会跨页)

```

1 initialise(worklist):
2     scan ← free
3     partialScan ← free
4
5 isEmpty(worklist):                                /* 与 Cheney 算法相同 */
6     return scan = free
7
8 remove(worklist):
9     if (partialScan < free)
10        ref ← partialScan                          /* 优先进行次级扫描 */
11        partialScan ← partialScan + size(partialScan)
12     else
13        ref ← scan                                  /* 主扫描 */
14        scan ← scan + size(scan)
15     return ref
16
17 add(worklist, ref):                                /* 在最近分配的页上进行次级扫描 */
18     partialScan ← max(partialScan, startOfPage(ref))

```

Moon 的算法最大缺陷在于, 它只记录了一对扫描指针, 无法区分指针 `scan` 和指针 `free` 之间的哪些对象已完成扫描, 因而有可能将某些对象扫描两次。Wilson 等 [1991] 声称 Moon 的算法重复扫描的比例大概有 30%, 并对此进行改进。他们为每一页记录指针 `scan` 和指针 `free`, 从而将工作列表变成所有需要进行部分扫描的块的链表, 因此主扫描可以跳过已经完成次级扫描的对象。

2.6 节曾讨论过如何提升标记-清扫回收器标记阶段的性能, 其中提到, Cher 等 [2004] 指出使用栈来引导的遍历遵从深度优先顺序, 但其对高速缓存行的预取却遵从广度优先顺序。因此一个自然而然的问题便是, 是否可以将基于栈的深度优先复制与 Cher 等 [2004] 的先进先出预取队列相结合? 很遗憾, 答案是否定的。尽管先进先出顺序可以减少高速缓存不命中对复制过程的影响, 但它会将父子节点分开, 因为只有当对象从预取队列中移除, 而不是从栈中移除时, 回收器才会访问对象所包含的引用[⊖]。我们来考察图 4.4 中将字符串对象 `S` 从栈中

⊖ 来自与 Tony Printezis 的私人交流。

弹出的过程：在理想情况下，对象 S 应当与其相关的字符数组 C 一起排列在目标空间中，这正是深度优先算法所能达到的效果。当使用先进先出队列时，S 从栈中弹出后将被立即添加到预取队列中，假设此时队列已满，则回收器会将最老的对象 X 从队列中移除并复制，同时将其引用的对象 Y 和 Z 压入栈中。但是，回收器从队列中移除和复制 Y 和 Z 将发生在 S 之后、C 之前。

上述各种复制算法的重排列方式都是静态的，即它们都没有考虑具体程序的实际行为，但可以肯定的是，对象重排列方式所能带来的收益最终取决于赋值器的行为。Lam 等 [1992] 发现，两种算法都对程序数据结构的组合方式以及形状十分敏感，对于非树形结构，其性能反而会有所降低。Siegwart 和 Hirzel [2006] 也发现，并行层次分解回收器可以提升某些基准测试程序的性能，但对于其他的则几乎没有效果。为解决这一问题，Huang 等 [2004] 对程序进行动态分析，并尝试将对对象的“热”域与其父节点排列在一起。算法 4.5 展示了他们的在线对象记录（online object recording）方法，图 4.2b 的最后一行展示了其效果。在算法的主扫描循环中（算法 4.5 中的第 6 行），回收器在对所有的“冷”域进行处理之前会先处理工作列表中的所有“热”域。对于一个配备了方法采样机制（method sampling mechanism）的自适应动态编译器而言，确定这些域的开销通常很小（Huang 等声称只占用不到 2% 的整体执行时间）。他们的算法也可以通过对“热”域的淘汰与重新扫描来适应程序在不同阶段的行为变化，他们发现在引入该算法后，系统的性能可以达到或者超过了诸如广度优先等静态重排列顺序。

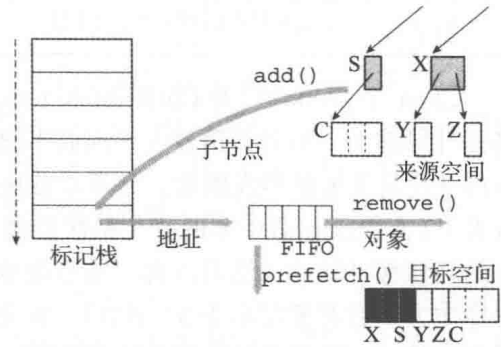


图 4.4 先进先出预取缓冲器（见第 2 章）无法提升赋值器局部性。它倾向于将“远亲”（C、Y、Z）而非父子节点排列在一起

算法 4.5 在线对象记录

```

1  atomic collect():
2      flip()
3      initialise(hotList, coldList)
4      for each fld in Roots
5          adviceProcess(fld)
6      repeat
7          while not isEmpty(hotList)
8              adviceScan(remove(hotList))
9          while not isEmpty(coldList)
10             adviceProcess(remove(coldList))
11     until isEmpty(hotList)
12
13 initialise(hotList, coldList):
14     hotList ← empty
15     coldList ← empty
16
17 adviceProcess(fld):
18     fromRef ← *fld
19     if fromRef ≠ null
20         *fld ← forward(fromRef)
21
22 adviceScan(obj):
23     for each fld in Pointers(obj)
24         if isHot(fld)

```

```

25         adviceProcess(fld)
26     else
27         add(coldList, fld)

```

Chen 等 [2006] 以及 Chilimbi 和 Larus[1998] 各自通过在分代回收器中主动调用回收器来提升局部性, 但其开销较大, 因而不会经常启用。对于以提升局部性为目标的回收, 分配率的变化是主要的触发因素, 转译后备缓冲区 (translation lookaside buffer, TLB) 中的数据或者 L2 高速缓存命中率的变化是次要触发因素。他们将得到访问的对象记录在一个固定大小的环状缓冲区中 (他们声称, 节点级别分析比域级别分析的开销要小 5%, 因为面向对象程序中大多数对象都小于 32 字节)。在突发式的采样过程中, 他们使用一个开销较大 (但经过高度优化) 的读屏障^①来拦截赋值器加载引用的操作, 并据此分辨热对象。热对象复制的过程分为两个阶段: 首先将赋值器正在访问的对象复制到一个临时缓冲区中, 然后使用层次分解的方法将热对象添加到该缓冲区以提升换页性能 [Wilson 等, 1991]。回收器将已复制对象的原有位置标记为空闲, 然后将临时缓冲区中经过重排列的对象移动到堆的一端。该方案尝试在高速缓存性能以及换页行为方面同时进行优化。实验结果表明, 将两种优化方法结合的收益通常大于两者各自收益的总和, 且对于多数大型 C# 应用程序而言, 平均执行时间都会得到改善。尽管该算法会保留部分垃圾对象, 但其总量通常很小。

还有学者提出, 可以依照对象类型来进行自定义的静态重排序 [Wilson 等, 1991; Lam 等, 1992], 特别是对于系统数据结构来说。通过允许类的开发者来决定域的复制顺序, Novark 等 [2006] 显著提升了某些特定数据结构的高速缓存命中率。Shuf 等 [2002] 使用离线分析 (off-line profiling) 的方法来确定富类型 (prolific type), 他们同时对分配器进行修改, 即当创建父对象时为其子对象预留相邻空间, 这样既提升了局部性, 同时又可以将具有相同生命周期的对象聚集在一起。对于本节所提到的将先进先出预取队列与深度优先复制相结合所带来的问题, 该方案可以在一定程度上予以缓解。

4.3 需要考虑的问题

相对于非移动式回收 (例如标记-清扫回收), 复制式回收具有两个显而易见的优点: 分配速度快, 同时可以根除内存碎片 (假设不考虑字节对齐要求)。简单的复制式回收器也比标记-清扫或者标记-整理回收器更容易实现, 但在相同的回收频率下, 复制式回收所需要的虚拟内存是其他回收器的两倍。

4.3.1 分配

在经过整理的堆中进行内存分配的速度很快, 其分配过程十分简单, 通常只需要简单判断堆或者内存块的上限, 然后返回空闲指针。如果使用块结构堆而非连续的堆, 则判断偶尔会失败, 此时则需使用一个新内存块。慢速路径^②的出现频率取决于所分配的对象平均大小与内存块大小的比例。在多线程情况下, 每个赋值器可以拥有一个独立的、无须与其他线程同步的本地分配缓冲区, 因而其分配速度也会很快。该方案实现简单, 且只需要很少的元数据, 相比之下, 如果非移动式回收器要使用本地分配策略, 每个线程可能需要一个独立的空

① 我们将在第 11 章讨论屏障技术。

② 即在判断失败情况下使用新内存块的代码路径。——译者注

间大小分级数据结构来实现分区适应分配。

阶跃指针分配的代码序列十分短小,更重要的是,这种线性分配方式具有较好的高速缓存友好性。在半区复制策略下使用顺序分配算法来分配短寿命对象,意味着下一个分配的位置很可能是最近最少使用的,但现代处理器的预取能力可以解决在这种问题下可能出现的时间延迟。如果这一行为与操作系统的最近最少使用(least recently used, LRU)页淘汰策略产生冲突进而影响到换页性能,那么就需要考虑重新配置系统。对于复制式回收而言,欲使程序运行流畅,要么需要更多的物理内存,要么需要借助于其他回收策略,例如第9章将介绍的分代垃圾回收器。

Blackburn等[2004a]发现,尽管在微型基准测试程序中顺序分配的性能比空闲链表分配要高11%,但是在真正的应用程序中,分配过程本身却只占用不到10%的整体执行时间,因此阶跃指针分配与空闲链表分配之间的开销差别就不那么显著了。分配过程只是赋值器所要完成的工作之一,而创建一个新对象的开销几乎是由其初始化方法决定的。在许多应用程序中,对象的生命周期都十分类似:赋值器几乎在同一时间创建一批语义相关的对象,使用它们,最后再一次性将它们全部丢弃。在这种情况下,经过整理的堆可以提供较好的局部性,它通常可以将相关对象分配在相同的页,如果对象较小,甚至可能在同一个高速缓存行。相对于从不同的空闲链表中进行分配的分区适应分配,顺序分配的高速缓存命中率更高。

4.3.2 空间与局部性

半区复制最显而易见的缺点是需要维护第二个半区,有时也称为复制保留区(copy reserve)。在内存大小一定,且忽略回收器所需数据结构的情况下,半区复制回收器的可用内存空间是整堆回收器的一半,这导致复制式回收器所需的回收次数比其他回收器更多。这一“交易”究竟会导致性能上的提升还是下降,取决于赋值器与分配器之间的平衡、应用程序的特征、可用堆空间的大小。

通过简单的渐进复杂度分析可以看出,复制式回收要优于标记-清扫回收。设 M 是堆的整体大小, L 为存活对象所占据的空间总量。半区复制回收器必须在存活数据中完成复制、扫描、更新指针的操作,标记-清扫算法同样必须遍历所有存活对象,但却需要扫描整个堆。Jones[1996]将复制式回收与标记-清扫回收的时间复杂度定义如下^①:

$$\begin{aligned} t_{\text{Copy}} &= cL \\ t_{\text{MS}} &= mL + sM \end{aligned}$$

每种回收器回收的内存量为:

$$\begin{aligned} m_{\text{Copy}} &= M/2 - L \\ m_{\text{MS}} &= M - L \end{aligned}$$

设 $r = L/M$,即存活数据比例,并假定其为固定值。算法的效率可以通过标记/构造率来描述,即赋值器每分配一个对象所对应的回收器工作量。我们用 e 表示标记/构造率,则两种算法的效率分别是:

$$e_{\text{Copy}} = \frac{2cr}{1-2r}$$

^① c 代表copy,即复制; m 代表mark,即标记; s 代表sweep,即清扫。——译者注

54

$$e_{\text{MS}} = \frac{mr + s}{1 - r}$$

通过图 4.5 中的标记 / 构造率曲线我们可以看出, 在堆足够大、 r 足够小的情况下, 复制式回收比标记 - 清扫回收更加高效, 但这一简单的分析方法忽略了几个因素。现代标记 - 清扫回收器通常使用懒惰清扫, 因此缩短了清扫时间 s , 降低了标记 / 构造率 e 。尽管 Herts 和 Berger[2005] 通过实验证实了曲线的整体形状 (例如标记 - 清扫回收的开销与堆大小成反比), 但是仍需谨慎对待复杂度的分析结果, 因为它忽略了具体的实现细节。对于真正的回收器而言, 具体的实现细节十分重要, 但这些细节并不能通过复杂度分析反映出来, 例如顺序分配所具有的局部性优势 [Blackburn 等, 2004a]。

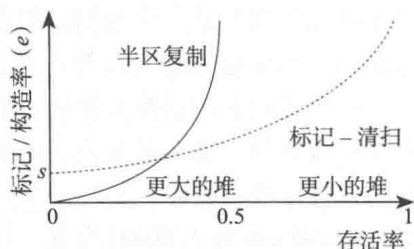


图 4.5 标记 - 清扫回收与复制式回收的标记 / 构造率 (越小越好)

虽然顺序分配有助于将同时受到赋值器访问的对象连续排列, 从而提升高速缓存命中率, 但是复制式回收却会将堆中存活对象重新排列。尽管 Cheney 回收器及其变种不需要辅助的栈来引导追踪, 但是其广度优先遍历顺序却有将父节点与子节点分开的趋势。层次分解策略在额外追踪栈的开销与对象排列方式的优化方面进行了折中, 尽管这一精心设计的重排列方式会使某些程序受益, 但其作用通常微不足道, 究其原因是因为大部分对象寿命较短, 通常活不过一次垃圾回收。此外, 许多应用程序的内存访问操作 (特别是写操作) 通常都集中在年轻的对象上 [Blackburn and McKinley, 2003], 回收器的遍历策略也无法影响被钉住的对象局部性。

Printezis 指出, 是否使用多线程并行回收也会影响复制机制的选择。与使用 Cheney 队列相比, 为每个线程使用独立的栈可以通过工作窃取 (work stealing) 策略实现更细粒度的负载均衡[⊖], 我们将在第 14 章进行详述。

4.3.3 移动对象

是否使用复制式回收器部分取决于移动对象的可行性及其开销。在某些环境下对象无法移动, 一方面是由于缺乏精确类型信息 (这意味着某个槽可能指向了一个“可能存在”的对象, 此时对该槽进行修改将是不安全的), 另一方面是由于对象被传递给了不允许引用发生变化的非托管代码 (例如作为系统调用的参数)。此外, 在标记 - 清扫环境下的指针寻找问题通常比移动式回收器下的简单。对于非移动式回收器, 找到指向某个存活对象的一个引用就已经足够, 但移动式回收器却一定要找到并且更新指向被移动对象的全部引用。正如我们将在第 17 章看到的, 这一问题在并发移动式回收器中更加严重, 因为指向某一对象的所有引用必须原子化地进行更新。

某些对象的复制开销很大, 即使对象所占用的空间很小, 对其进行复制的开销仍可能会远大于标记操作的开销, 但与此相比, 指针追踪以及获取对象类型信息的开销和延迟往往更大。另外, 一遍又一遍地复制不包含指针的大对象将会导致回收器性能的下降。一种解决方案是不复制大对象, 转而将其交由非移动式回收器管理; 另一种策略是使用虚拟的、非物理上的复制, 要达到这一目的可以将对象保存在由回收器维护的链表中, 或者将大对象分配在其专属的、可以二次映射 (remap) 的虚拟内存页上, 第 8 ~ 10 章将讨论这些技术。

55

56

⊖ 来自与 Tony Printezis 的私人交流。

引用计数

到目前为止，我们所介绍的垃圾回收算法都是间接式的，它们都需要从已知的根集合出发对存活对象图进行遍历，进而才能确定所有的存活对象。本章将介绍最后一种基本回收算法：引用计数（reference counting）[Collins, 1960]。在引用计数算法中，对象的存活性可以通过引用关系的创建或删除直接判定，从而无须像追踪式回收器那样先通过堆遍历找出所有的存活对象，然后再反向确定出未遍历到的垃圾对象。

引用计数算法所依赖的是一个十分简单的不变式：当且仅当指向某个对象的引用数量大于零时，该对象才有可能是存活的[⊖]。在引用计数算法中，每个对象都需要与一个引用计数相关联，这一计数通常保存在对象头部的某个槽中。算法 5.1 展示了最简单的引用计数实现，即当创建或者删除某一对象的引用时增加或者减少该对象的引用计数。Write 方法用于增加新目标对象的引用计数，同时减少旧目标对象的引用计数，即使对于局部变量的更新也需如此。我们同时假设，在一个方法返回之前，赋值器会将所有局部变量中的引用设置为空。addReference 方法实现对象引用计数的增加，相应地，deleteReference 实现引用计数的减少。需要注意的是，对引用计数的修改要遵循先增后减的顺序（算法 5.1 中的第 9 ~ 10 行），否则当新对象和老对象相同，也就是 `src[i]=ref` 时，可能导致对象被过早回收。一旦某一对象的引用计数降至零（算法 5.1 中的第 20 行），便可以将其回收，同时减少其所有子节点的引用计数，这可能引发子节点递归式的回收。

算法 5.1 简单的引用计数算法

```

1 New():
2   ref ← allocate()
3   if ref = null
4     error "Out of memory"
5   rc(ref) ← 0
6   return ref
7
8 atomic Write(src, i, ref):
9   addReference(ref)
10  deleteReference(src[i])
11  src[i] ← ref
12
13 addReference(ref):
14   if ref ≠ null
15     rc(ref) ← rc(ref) + 1
16
17 deleteReference(ref):
18   if ref ≠ null
19     rc(ref) ← rc(ref) - 1

```

⊖ 引用链（reference listing）算法对这一不变式进行了修改：当且仅当持有某一对象的客户端集合不为空时，才能判定对象存活。修改后的不变式具有一定的容错性，例如集合的插入和删除操作是幂等的而非算术的。该算法在分布式系统中应用广泛，例如 Java 的 RMI 库。

```
20     if rc(ref) == 0
21         for each fld in Pointers(ref)
22             deleteReference(*fld)
23         free(ref)
```

算法 5.1 中的 `Write` 方法是写屏障的一个例子，此处编译器在真正的指针写操作之外增加了一些简短的代码序列。我们将在后面章节看到，许多系统中的赋值器都需要执行一些额外的屏障操作。更确切地讲，如果不希望回收器关注整个对象图中所有对象的存活性，则赋值器必须承担一定的工作。此类回收器可能会与赋值器并发执行：要么在赋值器的引用计数操作中立即执行，要么在另一个线程中异步地执行。另外，回收器也可能以不同的频率来处理堆中不同区域的对象，例如分代式回收器。在这些情况下，赋值器必须引入一些额外的屏障操作来确保回收算法的正确性。

57

5.1 引用计数算法的优缺点

引用计数算法之所以能够成为一种有竞争力的自动内存管理策略，是由下面几个原因决定的。引用计数算法的内存管理开销分摊在程序运行过程中，同时一旦某一对象成为垃圾便可立即得到回收（但在后文我们将看到，这一特性并非对所有场合都有益），因此引用计数算法可以持续操作即将填满的堆，而不必像追踪式回收器那样需要一定的保留空间。引用计数算法直接操作指针的来源与目标，因此其局部性不会比它所服务的应用程序差。当应用程序确定某一对象并非共享对象时，可以直接对其进行破坏性的操作而无须事先创建副本。引用计数算法的实现无须运行时系统的支持，特别是无须确定程序的根。即使当系统部分不可用时，引用计数算法也能回收部分内存，这一特性在分布式系统中将是十分有用的 [Rodrigues and Jones, 1998]。

正是由于这些原因，引用计数算法在众多系统中得到广泛应用，包括一些编程语言实现（如早期的 Smalltalk 和 Lisp，以及 awk、perl 和 python）、部分应用程序（如 Photoshop，Real Networks 的 Rhapsody 音乐服务，Océ 打印、扫描及文档管理系统）以及操作系统的文件管理模块。对于 C++ 等尚不支持自动内存管理的语言，有许多支持对象安全回收的库，它们通常使用智能指针（smart pointer）来访问对象。智能指针通常会对构造函数以及赋值操作进行重载（overload），从而实现对象所有权的独占或者提供引用计数能力。唯一指针（unique pointer）能确保对象只会存在一个“所有者”，当对象的所有者被回收时，对象自身也将被回收。例如，下一代 C++ 标准预计会引入 `unique_ptr` 模版类^①。许多 C++ 开发者使用智能指针的引用计数能力来实现自动内存管理。C++ 中最著名的智能指针库当属 Boost 库^②，它通过共享指针对象来提供引用计数能力。智能指针的一个缺点是，尽管其看起来很像是原生指针，但是它们之间仍存在一些语义上的差异 [Edelson, 1992]。

58

但是，引用计数也存在一系列缺陷。第一，引用计数给赋值器带来了额外的时间开销。与前面章节中介绍的追踪式回收器相比，为管理引用计数，算法 5.1 对指针的 `Read` 和 `Write` 操作都进行了重定义，这将导致即使某些不具有破坏性的操作也必须付出一定的额外开销，例如在对链表进行迭代时，链表中每个对象的引用计数都需要先增加，接着减少一次。从性能角度来看，如果对寄存器或者线程栈上槽的操作也需要引入这一开销，显然是不能接受

① 目前 C++11 标准已经支持 `unique_ptr`。——译者注

② C++ Boost 库：<http://www.boost.org>。

的。因此仅这一项因素便决定了引用计数算法不适用于通用的大容量、高性能内存管理器，但幸运的是，我们可以通过一些方法来大幅降低引用计数操作的开销。

第二，为避免多线程竞争可能导致的对象释放过早，引用计数的增减操作以及加载和存储指针的操作都必须是原子化的，而仅对引用计数的增减操作进行保护是不够的。此处我们暂且不谈原子化的实现方式，只是简单地假设引用计数变更操作满足原子化要求，然后在第 18 章讨论引用计数和并发问题的细节。某些提供引用计数的智能指针库需要调用者小心使用以避免竞争。例如在 Boost 库中，对于一个 `shared_ptr` 实例，并发线程可以同时对其进行读取和修改操作，但库本身只能保证对引用计数的操作是原子化的，而智能指针的读写与引用计数增减这个整体过程却并非单独的原子操作。因此，开发者必须避免在更新指针槽过程中可能出现的竞争问题，否则将可能产生未定义的行为。

第三，在简单的引用计数算法中，即使是只读操作也需要引发一次内存写请求（用以更新引用计数）。类似地，对某一指针域进行修改时也需要对该域原本指向的对象进行读写操作各一次。这里的写操作会“污染”高速缓存，同时可能引发额外的内存冲突。

第四，引用计数算法无法回收环状引用数据结构（即包含自引用的数据结构）。即使此类数据结构在对象图中成为孤岛（即整体不可达时），其各个组成对象的引用计数也不会降至零。但是，自引用数据结构十分普遍（例如双向链表、存在从子节点指向根节点的指针的树等），尽管在不同程序中它们的出现频率相差较大 [Bacon and Rajan, 2001]。

第五，在最坏情况下，某一对象的引用计数可能等于堆中对象的总数，这意味着引用计数所占用的域必须与一个指针域的大小相同，即一个完整的槽。鉴于面向对象语言中对象的平均大小通常较小（例如 Java 程序中对象的大小通常是 20 ~ 64 字节 [Dieckmann and Holzel, 1999、2001; Blackburn 等, 2006a]），这一空间开销便显得十分昂贵。

最后，引用计数算法仍有可能导致停顿的出现。当删除某一大型指针结构根节点的最后一个引用时，引用计数算法会递归地删除根节点的每一个子孙节点。Boehm[2004] 声称，线程安全的引用计数回收所导致的最大停顿时间可能会比追踪式回收器的还要长。Weizenbaum[1969] 提出了一种懒惰引用计数（lazy reference counting）策略，即当某一对象的引用计数降至零时，`deleteReference` 不是立即将其回收，而是将其添加到一个待回收对象链表中，同时避免毁坏它的内容。当分配器重新将其分配出去时，才对其子节点进行处理，从而避免了递归释放问题。但这一技术会导致某些小型对象将大型垃圾对象隐藏，从而提升了整体空间需求 [Boehm, 2004]。

在引用计数算法面临的主要问题中，有两项可以得到解决，即引用计数操作的开销问题以及环状垃圾的回收问题。对于这两个问题的解决，一般都需要引入万物静止式的停顿，我们将在第 18 章探讨如何放宽这一限制。

5.2 提升效率

引用计数算法的效率可以从两方面进行提升：一方面是减少屏障操作的次数，另一方面是用更加廉价的非同步操作来替代昂贵的同步操作。Blackburn 和 McKinley[2003] 将各种解决方案分类如下：

延迟（deferral） 延迟引用计数（deferred reference counting）以牺牲少量细粒度回收增量的时效性（即当对象成为垃圾时立即将其回收）来换取效率的提升。该方案将某些垃圾对象的鉴别推迟到某一时段结束时的回收阶段中，从而避免了某些屏障操作。

合并 (coalescing) 许多引用计数操作都是临时性的、“不必要”的，开发者可以手动去掉这些无用操作，在某些特殊场景下编译器也可以完成这一工作，但更加通用的方法可能是在程序运行时仅跟踪对象在某一时段开始和结束时的状态。在单个时段内，合并引用计数 (coalescing reference counting) 只关注对象是否被第一次修改，针对同一对象的再次修改则会被忽略。

缓冲 (buffering) 缓冲引用计数 (buffered reference counting) 同样会延迟垃圾对象的鉴别。但与延迟引用计数或合并引用计数不同，该方案将所有的引用计数增减操作缓冲起来以便后续处理，同时只有回收线程可以执行引用计数变更操作。缓冲引用计数关注的是在“何时”执行引用计数变更操作，而不是“是否”需要进行变更。

引用计数算法在效率方面的一个基本障碍是：对象的引用计数是程序的全局特征之一，但通常只有在 (线程) 局部状态下的操作才更加高效^①。上述三种方案解决这一问题的思想都是相通的，即将程序的执行划分为一系列时段 (epoch)，在同一时段内赋值器可以省略部分甚至所有的同步引用计数操作，或者将其替换为 (在线程本地缓冲区上的) 非同步的写操作。垃圾的鉴定是在每个时段结束时进行的，此时便需要将赋值器线程挂起，或者使用一个 (或者多个) 独立的回收器线程与赋值器线程并发进行处理。不论对于哪种方案，单个时段内的本地引用计数状态变更均不会暴露到全局 (即：使用原子操作来将引用计数的状态变更作用到全局)。

本章将讨论延迟引用计数与合并引用计数。在这两种回收算法中，相邻回收时段会被万物静止式的停顿分开以实现引用计数的修正。第 18 章将介绍缓冲引用计数如何将引用计数操作转移给其他并发线程，以及如何并发地进行合并引用计数。

60

5.3 延迟引用计数

与简单的追踪式回收算法相比，引用计数操作给赋值器带来的开销相对较高。在后面章节中我们将看到，分代与并发回收算法也会给赋值器带来一定的开销，但其远远小于安全地进行引用计数操作所需的开销。算法 5.1 中的指针写操作需要多条指令才能实现 (尽管在某些条件下编译器可以静态优化掉某些判断)。引用计数的变更必须是原子化的且必须与指针的变更保持一致。另外，写操作对新老对象都要进行修改，从而很可能导致高速缓存被一些无法立即复用的数据污染。手工删除无用的引用计数操作很容易出错，而事实证明，编译器优化是解决这一问题的有效方案 [Cann and Oldehoeft, 1988]。

大多数高性能引用计数系统 (例如 Blackburn and McKinley[2003]) 都使用延迟引用计数策略。绝大多数指针加载操作都是将其加载到局部变量或者临时变量，即寄存器或者栈槽中。Deutsch 和 Bobrow[1976] 在很早以前就展示了如何移除这一情况下的引用计数操作，即只有当赋值器将指针写入堆中对象时才调整其目标对象的引用计数。图 5.1 展示了延迟引用



图 5.1 延迟引用计数示意图。其中展示了在进行指针加载或者存储时，赋值器是应当立即执行引用计数操作，还是将其延迟。箭头表示指针的资源和目标加载和存储的方向

Blackburn and McKinley[2003], doi: 10.1145/949305.94336.

© 2003 Association for Computing Machinery, Inc., 经许可后转载

① 意思是操作栈或者寄存器通常比操作堆中对象更加高效。——译者注

计数的抽象视图，只有当赋值器操作堆中对象时产生的引用计数变更才会立即执行，而操作栈或寄存器所产生的引用计数变更则会被延迟执行。这当然是要付出一定代价的：如果忽略局部变量的引用计数操作，则引用计数便不再准确，因此立即回收引用计数为零的对象便不再安全。为了确保所有的垃圾都能够得到回收，延迟引用计数必须引入万物静止式的停顿来定期修正引用计数，但幸运的是，这种停顿时间通常会比追踪式回收器用的时间短，例如标记-清扫回收器 [Ungar, 1984]。

在算法 5.2 中，赋值器加载对象所使用的读操作是第 1 章中所介绍的简单的、无屏障的实现方案，将引用写入根的操作也是无屏障的（见算法 5.2 中的第 14 行），但将引用写入堆中对象时却必须使用屏障，在这种情况下必须立即增加新对象的引用计数（见算法 5.2 中的第 17 行）。当某一对象的引用计数变为零时，写屏障需要将其添加到零引用表（zero count table, ZCT）中，而不能立即将其释放（见算法 5.2 中的第 26 行），因为程序栈中仍可能存在该对象的引用，但该引用并未计入到对象的引用计数中。零引用表可以通过多种方式实现，例如位图 [Baden, 1983] 或者哈希表 [Deutsch and Bobrow, 1976]。从概念上讲，零引用表中包含的对象都是引用计数为零但可能依旧存活的对象。当赋值器把某一零引用对象的引用写入堆中对象时可以将其从零引用表中移除，因为此时该对象的引用计数必然为一个正数（见算法 5.2 中的第 19 行），这一策略有助于控制零引用表的大小。

当堆可用内存耗尽时（例如分配器无法分配内存）就必须进行垃圾回收。回收器需要挂起所有赋值器线程并检查零引用表中对象的引用计数是否真正为零。对于零引用表中的对象，只有当其被一个或者多个根引用时，该对象才可以被确定是存活的。确定零引用表中存活对象的最简单方法是对根所指向的对象进行扫描，并增加其引用计数（见算法 5.2 中的第 29 行），这一步完成后，所有被根引用的对象的引用计数必然都为正数，而引用计数为零的对象则是垃圾。为实现垃圾对象的回收，我们可以采用与标记-清扫类似的方法（例如算法 2.3）扫描整个堆，即找到且回收所有引用计数为零的“未标记”对象，但仅扫描零引用表也能达到相同的效果，即采用与算法 5.1 类似的方法来处理并释放零引用表中的对象。最后，必须还原“标记”操作，即再次对根进行扫描，并将其目标对象的引用计数减 1（即将引用计数恢复到其原有的值）。此时如果某个对象的引用计数再次归零，则需重新将其加入零引用表。

延迟引用计数消除了赋值器操作局部变量时的引用计数变更开销。一些较早的研究表明，延迟引用计数可以将指针操作的开销减少 80% 甚至更多 [Ungar, 1984; Baden, 1983]，如果再考虑其对局部性的提升，那么在现代硬件条件下，其在性能提升方面应该更具优势。然而，对象指针域的引用计数操作却无法延迟，而必须立即执行，且必须为原子操作。针对这一问题，我们将在接下来的一节中探讨，具体包括如何使用简单的方法替代由对象域变更引起的昂贵的引用计数原子操作，以及如何减少引用计数的修改次数。

算法 5.2 延迟引用计数

```

1 New():
2   ref ← allocate()
3   if ref = null
4     collect()
5     ref ← allocate()
6   if ref = null
7     error "Out of memory"
8   rc(ref) ← 0

```

```

9      add(zct, ref)
10     return ref
11
12 Write(src, i, ref):
13     if src == Roots
14         src[i] ← ref
15     else
16         atomic
17             addReference(ref)
18             remove(zct, ref)
19             deleteReferenceToZCT(src[i])
20             src[i] ← ref
21
22 deleteReferenceToZCT(ref):
23     if ref ≠ null
24         rc(ref) ← rc(ref) - 1
25         if rc(ref) = 0
26             add(zct, ref) /* 延迟释放 */
27
28 atomic collect():
29     for each fld in Roots /* 标记栈 */
30         addReference(*fld)
31     sweepZCT()
32     for each fld in Roots /* 反标记栈 */
33         deleteReferenceToZCT(*fld)
34
35 sweepZCT():
36     while not isEmpty(zct)
37         ref ← remove(zct)
38         if rc(ref) = 0 /* 执行垃圾回收 */
39             for each fld in Pointers(ref)
40                 deleteReference(*fld)
41                 free(ref)

```

5.4 合并引用计数

延迟引用计数解决了赋值器操作局部变量时的引用计数变更开销，但是当赋值器将某一对象的引用存入堆时，引用计数的变更开销依然无法避免。Levanoni 和 Petrank[1999] 注意到，对于任意时段内的任意对象域，回收器只需关注其在该时段开始和结束时的状态，而时段内的引用计数操作则可以忽略，因此可以将对象的多个状态合并成两个。例如，假设初始状态下对象 X 的指针域 f 引用了对象 O_0 ，该域在某个时段内依次被修改为 O_1, O_2, \dots, O_n ，此时引用计数的更新操作如下所示：

$$rc(O_0)--, \boxed{rc(O_1)++, rc(O_1)--}, \boxed{rc(O_2)++, \dots}, rc(O_n)++.$$

中间状态的一对操作（见实线框内）相互抵消了，因而可以省略。Levanoni 和 Petrank 的方法是，在每个时段内，写屏障会在对象首次得到修改之前将其复制到本地日志中。对于在当前时段尚未修改过的对象，当赋值器更新其某一指针域时，算法 5.3 会捕获这一操作并将对象的地址及其每个指针域的值都记录到本地更新缓冲区中（算法 5.3 中的第 5 行），同时将被修改的对象标记为脏。

在 log 方法中，为避免将对象重复加入线程本地日志，算法先将对象指针域的初始值添加到日志中（算法 5.3 中的第 11 行），同时只有当 `src` 不为脏时才将其添加到日志中（`appendAndCommit` 方法），然后再增加日志的内部游标（算法 5.3 中的第 13 行），此时算法需

要将对象打上脏标记以便与指针域进行区分。将对象标记为脏的方法是将其在日志中对应条目的地址写入其头域。需要注意的是，即使竞争导致在多个线程本地缓冲区中出现同一对象的条目，算法也能保证各个条目包含相同的信息，因此也无须关心对象头域中所记录的日志条目究竟位于哪个线程的本地缓冲区中。基于处理器的内存一致性模型，写屏障很可能不需要任何同步操作。

算法 5.3 合并引用计数：写屏障

```

1  me ← myThreadId
2
3  Write(src, i, ref):
4      if not dirty(src)
5          log(src)
6          src[i] ← ref
7
8  log(obj):
9      for each fld in Pointers(obj)
10         if *fld ≠ null
11             append(updates[me], *fld)
12         if not dirty(obj)
13             slot ← appendAndCommit(updates[me], obj)
14             setDirty(obj, slot)
15
16  dirty(obj):
17      return logPointer(obj) ≠ CLEAN
18
19  setDirty(obj, slot)
20      logPointer(obj) ← slot      /* 对象 obj 在 updates[me] 中对应条目的地址 */

```

本章中我们将简单地使用万物静止式停顿来周期性地处理日志，而对于如何在赋值器线程执行的同时并发处理合并引用计数，将在后面章节中讨论。在回收周期的开始阶段，算法 5.4 先将每个线程挂起，然后将每个线程的更新缓冲区合并到回收器的日志中，最后再为每个线程分配新的更新缓冲区。上文提到，竞争关系可能导致多个线程的更新缓冲区中包含同一对象的条目，这就要求回收器确保对每个脏对象只会进行一次处理，因此 processReferenceCounts 方法在更新引用计数之前会先判断对象是否为脏。对于标记为脏的对象，回收器先清空其脏标记以确保不会对它进行重复处理，然后将回收时刻[⊖]其所有子节点的引用计数加 1，最后再将当前时段内该对象首次得到修改之前的子节点[⊖]的引用计数减 1。在简单的引用计数系统中，一旦某一对象的引用计数降至零则会立即得到递归释放，但如果算法将局部变量的引用计数变更延迟，或者出于效率原因无法确保所有的引用计数的增加操作先于减少操作执行，则需要将零引用对象记录到零引用表中。在该时段内，对象的最初子节点可以直接从日志中获得，而对象当前的子节点则可以从对象自身直接获取（日志中包含了该对象的引用）。另外，在增加和减少引用计数的循环中，算法都可以进行对象或者引用计数域的预取 [Paz and Petrank, 2007]。

算法 5.4 合并引用计数：更新引用计数

```

1  atomic collect():

```

⊖ 即当前时刻，时段结束时刻。——译者注

⊖ 即该对象在时段开始时的子节点。——译者注

```

2   collectBuffers()
3   processReferenceCounts()
4   sweepZCT()
5
6   collectBuffers():
7       collectorLog ← []
8       for each t in Threads
9           collectorLog ← collectorLog + updates[t]
10
11  processReferenceCounts():
12      for each entry in collectorLog
13          obj ← objFromLog(entry)
14          if dirty(obj)                                /* 避免重复处理 */
15              logPointer(obj) ← CLEAN
16              incrementNew(obj)
17              decrementOld(entry)
18
19  decrementOld(entry):
20      for each fld in Pointers(entry)                  /* 使用回收器日志中的值 */
21          child ← *fld
22          if child ≠ null
23              rc(child) ← rc(child) - 1
24              if rc(child) = 0
25                  add(zct, child)
26
27  incrementNew(obj):
28      for each fld in Pointers(obj)                    /* 使用对象中的值 */
29          child ← *fld
30          if child ≠ null
31              rc(child) ← rc(child) + 1

```

65

我们以图 5.2 为例来演示合并引用计数的处理过程。假设对象 A 的某个指针域在某一时段内从对象 C 修改为对象 D，则在该时段结束时，对象 A 的两个指针域原有的值（B 和 C）则已经记录在回收器日志中（图 5.2 的左边）了，因此回收器会增加对象 B 和对象 D 的引用计数，同时减少对象 B 和 C 的引用计数。由于对象 A 中指向对象 B 的指针域并未修改，因此对象 B 的引用计数不变。

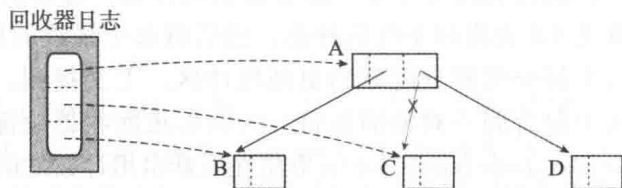


图 5.2 合并引用计数：假设对象 A 在上一个时段内被修改，例如将指向对象 C 的引用修改为指向对象 D，则 A 的引用域会被记录到日志中。原本指向对象 C 的引用可以从日志中找到，而当前指向对象 D 的引用可以直接从对象 A 中获取

将延迟引用计数与合并引用计数相结合可以降低赋值器上大部分引用计数操作的开销。特别需要指出的是，我们通过这两种方法消除了赋值器线程对昂贵的同步操作的依赖，但这些收益也是要付出一定代价的。为了进行垃圾回收，我们要再次引入了停顿，尽管这一停顿的时间可能会比追踪式回收器短。我们降低了回收的时效性（垃圾对象只有在时段结束时刻才能得到回收），同时日志缓冲区与零引用表也带来了额外的空间开销。在合并引用计数中，对于一个从未得到修改的指针槽，它所指向的对象仍有可能需要回收器各增删引用计数一次[⊖]。

⊖ 如图 5.2 中的对象 B。——译者注

5.5 环状引用计数

对于环状数据结构而言,其内部对象的引用计数至少为 1,因此仅靠引用计数本身无法回收环状垃圾。不论是在应用程序还是在运行时系统中,环状数据结构都十分普遍,如双向链表或者环状缓冲区。对象-关系映射(object-relations mapping)系统可能要求数据库和其中的表互相引用对方的一些信息。真实世界中的某些结构天然就是环状的,例如地理信息系统中的道路。懒惰函数式语言(lazy functional language)通常使用环来表示递归[Turner, 1979, Y 组合子(Combinator)]。研究者们提出了多种解决环状引用计数问题的策略,我们介绍其中的几种。

最简单的策略是在引用计数之外偶尔使用追踪式回收作为补充。该方法假定大多数对象不会被环状数据结构所引用,因此可以通过引用计数方法实现快速回收,而追踪式回收则负责处理剩余的环状数据结构。这一方案简单地减少了追踪式回收的发起频率。在语言层面上,Friedman 和 Wise[1979]发现,纯函数式语言中只有递归定义才会产生环,因此只要遵从一定的规则便可对这种情况下的环状引用计数进行特殊处理。在 Bobrow[1980]的方法中,开发者可以把一组对象作为整体进行引用计数操作,当整体引用计数为零时便可将其集体回收。

66

许多学者建议将导致闭环出现的指针与其他指针进行区分[Friedman and Wise, 1979; Brownbridge, 1985; Salkild, 1987; Pepels 等, 1988; Axford, 1990]。他们将普通引用称为强引用(strong reference),将导致闭环出现的引用称为弱引用(weak reference)。如果不允许强引用组成环,则强引用图可以使用标准引用计数算法处理。Brownbridge 的算法得到了广泛应用,简而言之,每个对象需要包含一个强引用计数以及一个弱引用计数,在进行写操作时,写屏障会检测指针以及目标对象的强弱,并将所有可能产生环的引用设置为弱引用。为维护“所有可达对象均为强可达,且强引用不产生环”这一不变式,赋值器在删除引用时可能需要改变指针的强弱属性。但是,这一算法并不安全,且可能导致对象提前被回收,具体可以参见 Salkild 的引用计数示例[Jones, 1996, 6.5 节]。Salkild[1987]对该算法进行修正并提升了它的安全性,但代价是在某些情况下算法将无法结束。Pepels 等[1988]提出了一种非常复杂的解决方案,但该算法在空间以及性能方面的开销却更加明显:与普通的引用计数相比,其所需的空间开销翻倍,在大多数情况下,其性能开销是标准引用计数的两倍,在极端情况下,甚至会呈现指数级增长。

在所有能够处理环状数据结构的引用计数算法中,得到了最广泛认可的是试验删除(trial deletion)算法。该算法无须使用后备的追踪式回收器来进行整个存活对象图的扫描,相反,它将注意力集中在可能会因删除引用而产生环状垃圾的局部对象图上。在引用计数算法中:

- 在环状垃圾指针结构内部,所有对象的引用计数均由其内部对象之间的指针产生。
- 只有在删除某一对象的某个引用后该对象的引用计数仍大于零时,才有可能出现环状垃圾。

部分追踪(partial tracing)算法充分利用上述两个结论,该算法从一个可能是垃圾的对象开始进行子图追踪。对于遍历到的每个引用,算法将对其目标对象进行试验删除,即临时性地减少目标对象的引用计数,从而移除由内部指针产生的引用计数。追踪完成后,如果某个对象的引用计数仍然不是零,则必然是因为子图之外的其他对象引用了该对象,进而可以

判定该对象及其传递闭包都不是垃圾。

Recycler 算法 [Bacon 等, 2001 ; Bacon and Rajan, 2001 ; Paz 等, 2007] 支持环状引用计数的并发回收。算法 5.5 仅演示了该算法较为简单的同步版本, 异步回收的版本则在第 15 章进行讨论。环状数据结构的回收分为 3 个阶段:

首先, 回收器从某个可能是环状垃圾成员的对象出发进行子图追踪, 同时减少由内部指针产生的引用计数 (markCandidates 方法)。算法将遍历到的对象着为灰色。

其次, 对子图中的所有对象进行检测, 如果某一对象的引用计数不是零, 则该对象必然被子图外的其他对象引用。此时需要对第一阶段的试验删除操作 (scan 方法) 进行修正, 算法将存活的灰色对象重新着为黑色, 同时将其他灰色对象着为白色。

最后, 子图中所有依然为白色的对象必然是垃圾, 算法可以将其回收 (collectCandidates 方法)。

67

算法 5.5 Recycler 算法

```

1 New():
2   ref ← allocate()
3   if ref = null
4     collect()                                /* 环状引用回收器 */
5     ref ← allocate()
6     if ref = null
7       error "Out of memory"
8   rc(ref) ← 0
9   return ref
10
11 addReference(ref):
12   if ref ≠ null
13     rc(ref) ← rc(ref) + 1
14     colour(ref) ← black                    /* 不可能在环状垃圾中 */
15
16 deleteReference(ref):
17   if ref ≠ null
18     rc(ref) ← rc(ref) - 1
19     if rc(ref) = 0
20       release(ref)
21   else
22     candidate(ref)                        /* 可能是一个环状垃圾 */
23
24 release(ref):
25   for each fld in Pointers(ref)
26     deleteReference(fld)
27   colour(ref) ← black                    /* 空闲链表中的对象均为黑色 */
28   if not ref in candidates                /* 备选垃圾将稍后处理 */
29     free(ref)
30
31 candidate(ref):                          /* 将 ref 标记为备选垃圾, 并将其添加到集合中 */
32   if colour(ref) ≠ purple
33     colour(ref) ← purple
34   candidates ← candidates ∪ {ref}
35
36 atomic collect():
37   markCandidates()
38   for each ref in candidates
39     scan(ref)
40   collectCandidates()
41 markCandidates()

```

```

42  for ref in candidates
43      if colour(ref) = purple
44          markGrey(ref)
45      else
46          remove(candidates, ref)
47          if colour(ref) = black && rc(ref) = 0
48              free(ref)
49
50  markGrey(ref):
51      if colour(ref) ≠ grey
52          colour(ref) ← grey
53      for each fld in Pointers(ref)
54          child ← *fld
55          if child ≠ null
56              rc(child) ← rc(child) - 1          /* 试验删除 */
57              markGrey(child)
58
59  scan(ref):
60      if colour(ref) = grey
61          if rc(ref) > 0
62              scanBlack(ref)                      /* 必然存在外部引用 */
63          else
64              colour(ref) ← white                  /* 像是垃圾 ... */
65              for each fld in Pointers(ref)        /* ... 继续 */
66                  child ← *fld
67                  if child ≠ null
68                      scan(child)
69
70  scanBlack(ref):                                /* 修正存活对象的引用计数 */
71      colour(ref) ← black
72      for each fld in Pointers(ref)
73          child ← *fld
74          if child ≠ null
75              rc(child) ← rc(child) + 1          /* 反向试验删除 */
76              if colour(child) ≠ black
77                  scanBlack(child)
78  collectCandidates():
79      while not isEmpty(candidates)
80          ref ← remove(candidates)
81          collectWhite(ref)
82
83  collectWhite(ref):
84      if colour(ref) = white && not ref in candidates
85          colour(ref) ← black                    /* 空闲链表中的对象为黑色 */
86      for each fld in Pointers(ref)
87          child ← *fld
88          if child ≠ null
89              collectWhite(child)
90      free(ref)

```

同步模式下的 Recycler 算法使用五种颜色来区分对象。与其他算法一样，黑色代表存活，白色代表垃圾，而灰色代表对象可能是环状垃圾中的一个成员，紫色表示对象可能是环状垃圾的一个备选根。

如果删除指向某一对象的一个引用之后其引用计数依然不是零，则可能导致环状垃圾的产生，因此算法 5.5 将其标记为紫色，同时将其添加到环状垃圾备选成员集合中（算法 5.5 中的第 22 行）。另外，如果删除指向某一对象的一个引用后其引用计数降至零，则该对象必

然是垃圾，release 方法会将其修改为黑色，并递归地处理其子节点。此时如果该对象不是备选垃圾，release 方法会直接将其释放，而如果对象包含在 candidates 集合中，对该对象的回收将会推迟到 markCandidates 阶段。如图 5.3a 中，当删除某一指向对象 A 的引用后，对象 A 的引用计数仍然不是零，此时会将该对象添加到 candidates 集合中。

在回收的第一阶段，markCandidates 方法首先限定可能是环状垃圾的对象的范围，并消除内部引用对引用计数的影响。该阶段会检测 candidates 集合中的每个对象。如果对象依然是紫色（即在该对象被添加到 candidates 集合之后，再没有新的引用指向该对象），则将其递归闭包都标记为灰色，否则便将其从集合中移除。如果对象为黑色且引用计数为零，则立即将其回收。markGrey 在追踪过程中会将其所遍历到的每个对象的引用计数减 1。因此在图 5.3b 中，从 A 开始的子图已被着为灰色，且由于子图内部引用所产生的引用计数都已经得到消除。

在回收的第二阶段，算法会对备选垃圾及其灰色传递闭包进行扫描，同时找出哪些对象存在外部引用。如果某一对象的引用计数不是零，则其必然受到灰色子图之外的某个对象的引用，在这种情况下，scanBlack 会对由 markGrey 造成的引用计数减少操作进行补偿，即增加对象的引用计数并将其着为黑色；如果对象的引用计数为零，则将其着为白色，并继续扫描其子节点。需要注意的是，这里不能将白色对象与垃圾等价，因为如果 scanBlack 方法从另一个节点开始进行子图遍历，则有可能再次访问到白色对象。如图 5.3b 中，尽管对象 Y 和 Z 的引用计数均为零，但它们依旧通过外部对象 X 可达。当 scan 方法遍历到对象 X 时会发现它的引用计数不是零，此时算法会调用 scanBlack 方法来修正其灰色传递闭包中对象的引用计数。子图最终的状态如图 5.3c 所示。

最后，在回收的第三阶段，collectWhite 方法将回收白色（垃圾）对象。该方法将清空 candidates 集合，同时将遍历到的每个白色对象释放（并将其重置为黑色），然后再递归处理其子节点。需要注意的是，collectWhite 方法不会处理已经存在于 candidates 集合中的子对象，它们将在 collectCandidates 方法内的后续循环中得到处理。

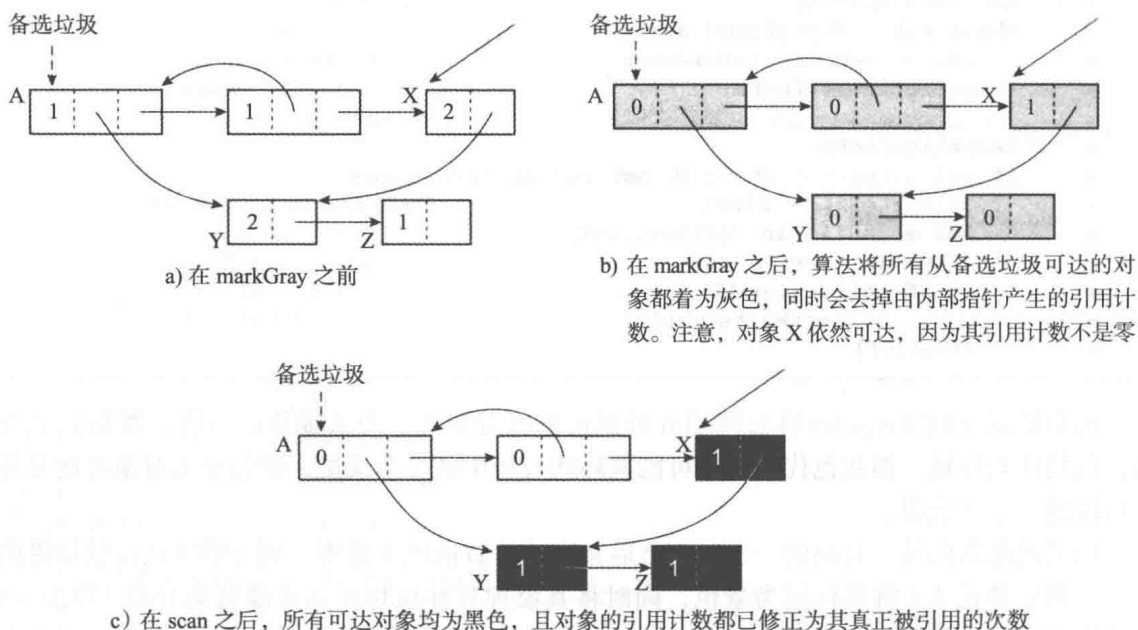


图 5.3 环状引用计数（每个对象的第一个域为其引用计数）

异步 Recycler 算法基于早期的试验删除算法改进而来, 例如 Martinez 等 [1990] 的算法, 该算法在发现备选垃圾时会立即进行试验删除。Lins[1992] 采用与 Recycler 算法类似的方法来对备选垃圾进行懒惰处理, 该算法期望赋值器能在后续操作中消除备选垃圾, 即要么赋值器会删除指向备选垃圾的最后一个引用使其立即得到回收, 要么会为其增加一个新的引用^①。但 Lins 的算法需要为每个对象分别进行三个阶段的回收处理, 在最差情况下, 算法的复杂度将与对象图大小的平方成正比, 相比之下, Recycler 算法的复杂度是 $O(N + E)$, 其中 N 是节点数量, E 为边数量。这一看起来很小的区别却会在性能方面造成很大的差异, 对于中等大小的 Java 程序, Lins 算法最大可能需要数分钟级的回收时间, 而 Recycler 算法则只需要毫秒级别的回收时间。

对某些类型的对象进行特殊处理可以进一步提升回收性能。此类对象包括不包含指针的对象、永远不可能是环状数据结构成员的对象等。Recycler 算法在分配这些对象时会将其着为绿色而非黑色, 同时决不会将其加入备选垃圾集合中, 更不会对其进行追踪。Bacon 和 Rajan[2001] 发现这一方法可以将备选垃圾集合的大小降低一个数量级。图 5.4 描述了同步 Recycler 算法完整的对象状态转化, 包括绿色节点在内。

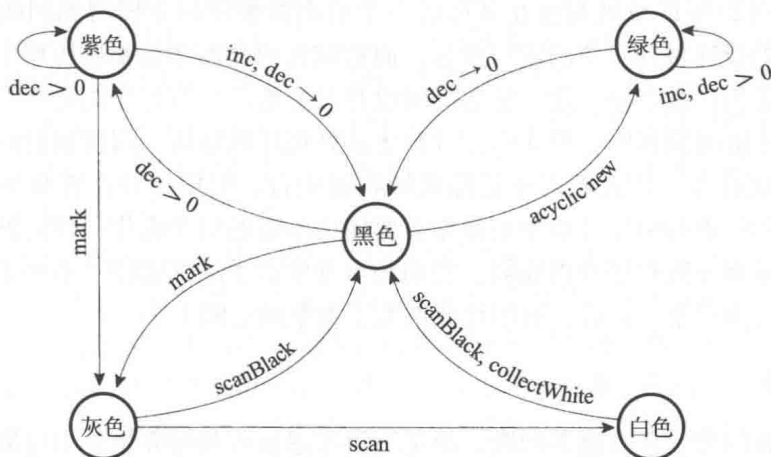


图 5.4 同步 Recycler 算法的状态转换图, 展示了赋值器和回收器的各项操作以及对象的颜色

Springer Science+Business Media: Bacon and Rajan [2001], 图 3, 第 213 页, 经许可后转载

5.6 受限域引用计数

对象的引用计数在其头部所占用的空间也是值得注意的。从理论上讲, 某一对象可能会被堆中所有的对象引用, 因此引用计数域的大小应当与指针域的大小相同, 但对于小对象而言, 这一级别的空间开销显得太过昂贵。在实际应用中, 大部分对象的引用计数通常较小, 除非是有意为之才会变得很大。另外, 大部分对象都不是共享对象, 一旦指向它们的指针被删除, 这些对象便可立即得到复用 [Clark and Green, 1977; Stoye 等, 1984; Hartel, 1988], 这一特性允许函数式语言对诸如数组等对象进行原地更新, 而不必基于其新的副本进行修改。如果事先知道引用计数可能达到的上限, 则可以使用较小的域来记录引用计数, 但许多程序中通常都会存在一些被广泛引用的对象 (popular object) [Printezis and

^① 从而使其不再可能是备选垃圾。——译者注

Garthwaite, 2002]。

在面对引用计数偶尔超出上限的问题时,如果能够引入后备处理机制,则仍有可能限制引用计数域的大小。一旦某个对象的引用计数达到允许的最大值,可以将其转变成粘性引用计数(sticky reference count),即之后的任何指针操作都不再改变该对象的引用计数值。最极端的选择是仅使用一个位来表示引用计数,从而将引用计数的能力集中在非共享对象上。该位可以保存在对象中[Wise and Friedman, 1977],也可以记录在指针上[Stoye等, 1984]。受限域引用计数的必然结果是:一旦对象的引用计数超出上限,则不能再通过引用计数来回回收对象,此时就需要后备的追踪式回收器来处理这种对象。追踪式回收器在遍历每个指针时可以将对象的引用计数修复到正确的值(不论该对象的引用计数是否超限)。Wise[1993a]表示,标记-整理和复制式回收器经过适当改造也能恢复对象的唯一性信息。后备的追踪式回收器应当在任何情况下都能回收环状垃圾。

5.7 需要考虑的问题

引用计数算法的回收时效性较高,且具有较好的局部性,因而具有一定的吸引力。简单的引用计数算法可以保证垃圾对象在其最后一个引用被删除时立即得到回收。引用计数的变更只会引发新老指针目标对象的读写操作,而追踪式回收器却需要遍历堆中所有的存活对象,但对于简单引用计数来说,这一优势同时也是其劣势。只有在指向对象的最后一个指针被删除时,对象才能得到回收,因而引用计数无法处理环状垃圾。每次指针读写操作都需要伴随引用计数变更操作,因此相对于追踪式回收器而言,引用计数在吞吐量方面的开销更大。多线程应用程序中的引用计数变更操作需要使用昂贵的同步操作。赋值器和内存管理器之间的紧耦合关系将导致程序变得脆弱,特别是当开发者手动优化掉“不必要”的引用计数时,这一情况将更加严重。最后,引用计数增大了对象的空间大小。

5.7.1 应用场景

尽管引用计数的方法存在诸多问题,但是不经考虑就将其抛弃也是不可取的。诚然,简单的引用计数算法中所存在的缺陷决定了它并不是虚拟机中通用内存管理模块的有力竞争者,特别是在对象较小、环状引用较为普遍、指针赋值率较高的场景下。但引用计数方法在某些场景下依然十分有用。在大多数对象的生命周期简单到可以直接进行管理的混合场景,引用计数算法可以较好地发挥功效。引用计数算法可以用于管理数量较少但所有者关系复杂的资源。这些资源通常较大,因此头部中引用计数域的额外开销通常可以忽略。某些数据结构并不包含指针,因此也不会出现环状引用,例如图像中的位图。另外,引用计数算法可以在代码库中实现,从而无需作为语言的运行时系统的一部分,因此开发者可以完全控制引用计数的使用,进而可以自主决定性能开销与安全性之间的平衡。但不论如何,使用引用计数时都应当小心谨慎。对于多线程环境下的指针修改以及引用计数更新中可能存在的竞争,开发者必须能避免。如果通过智能指针来实现引用计数,则开发者必须明确它与原生指针在语义上的差异,正如Edelson[1992]所指出的:“它们虽然智能,但并非指针”。

5.7.2 高级的解决方案

高级引用计数算法可以解决原生引用计数算法中存在的诸多问题,但矛盾的是,这些算法却需要引入与追踪式回收类似的万物静止式停顿。下一章将进一步讨论这一具有两面性的

问题。

正如 5.5 节所描述的，环状垃圾可以由后备的追踪式回收器或者试验删除算法来处理，但这两种策略都需要在回收环状数据时挂起赋值器线程（后面章节将描述如何避免这种万物静止式的停顿）。

在最坏的情况下，引用计数域的大小几乎要与指针域的大小相等，但在大部分应用程序中，大多数对象的被引用次数通常较少。引用计数通常可以复用对象现有头部字中的几个位（例如与对象哈希或者锁共用同一个）。但极少数对象被广泛引用的情况也经常发生。如果使用受限域引用计数可能需要后备的追踪式回收器来处理这些对象，但如果出现引用计数溢出的对象数量较少，且其生命周期较长，可以容许这些对象产生泄漏。仅与标记 - 清扫回收进行比较并不能简单地断定引用计数算法的空间开销更大，因为追踪式回收器必须在堆中保留一定的空间来避免性能上的颠簸，如果要求堆空间比最大存活对象总大小还要大 20%，则意味着至少 10% 的空间将被浪费，这一比率很可能与引用计数算法的空间开销相近（取决于对象的平均大小）。

省略某些引用计数操作可以降低引用计数算法在吞吐量方面的开销。延迟引用计数会忽略赋值器对局部变量的操作，这将导致直接从根可达的对象的引用计数比其真实值要小，进而损失了回收的时效性（因为引用计数为零不再意味着对象就是垃圾）。合并引用计数仅关注对象在某一时段开始和结束时的状态，同时忽略时段内的指针操作。从某种意义上讲，这一方法是将开发者手动优化临时性引用计数变更的工作自动化（例如用于遍历链表的迭代器）。然而，使用延迟引用计数和合并引用计数的一个后果是，为了修正引用计数，它们再次引入了万物静止式的停顿。

使用延迟引用计数和合并引用计数不仅可以省略某些引用计数操作，而且可以减少其他操作的同步开销。延迟引用计数简单地忽略了针对局部变量的引用计数操作；在合并引用计数中，竞争是良性的，因此无需进行同步，最差的情况是可能将相同的值写入两个不同线程的日志中。然而，这两种策略都引入了额外的空间开销，例如零引用表以及更新日志。

74

这些高级引用计数方案的另一个优势在于它们可以处理规模较大的堆，其处理开销仅与指针操作的次数相关，而与存活对象的大小无关。我们将在第 10 章看到，追踪式回收器与引用计数算法可以组合成混合式回收器，其中前者将用于处理短命的、操作频繁的对象，后者将用于处理长寿的、更加稳定的对象。

第 6 章将对已经介绍过的 4 种回收算法进行比较，它们分别是：标记 - 清扫、标记 - 整理、复制式，以及引用计数。届时我们将对追踪式回收以及高级引用计数回收进行高度抽象，并揭示它们在某些方面的惊人相似性。

75
76

垃圾回收器的比较

前面介绍了 4 种不同类型的垃圾回收器，本章将对它们进行更加详细的比较。我们将从两个不同方面进行考察：首先我们会确立一套标准来评价不同回收算法在不同场景下的优势与不足；然后再介绍 Bacon 等 [2004] 对追踪式回收算法和引用计数算法的抽象。我们将看到，虽然不同算法在表面上存在差异，但它们在更深层次上却存在着显著的相似性。

读者通常会问：究竟哪种垃圾回收器最好？但这一问题并不存在简单的答案。首先需要明确“最好”的定义是什么：是希望应用程序的吞吐量最大，还是希望回收的停顿时间最短？是希望空间使用率最高，还是希望对这些参数进行综合考虑以达到整体上的平衡？其次，在不同的应用程序中，即使对于同一个评价标准，不同回收器的排名也可能有所不同。例如，Fitzgerald 和 Tarditi[2000] 通过对 20 种 Java 基准测试程序和 6 种不同的回收器进行研究发现，对于任意一种回收器，都存在另一种更加合适的回收器可以将某个特定基准测试程序的执行速度提升至少 15%。对于不同大小的可用堆，回收器的性能也各不相同。如果可用堆空间更大，则程序通常执行得更快，但如果可用堆空间过大，则相关对象在空间上更加分散，程序的空间局部性更低，进而可能降低程序的性能。这些因素都使得问题的分析更加复杂。

6.1 吞吐量

对于许多用户而言，他们关注的首要问题可能是程序的整体吞吐量，这同时也可能是批处理程序或者网络服务器的主要评价指标。对于前者而言，短暂的停顿可以接受，而对于后者，这种停顿往往会被系统或者网络的延迟所掩盖。尽量快地执行垃圾回收固然重要，但更快的回收速度并不意味着程序整体执行速度也会更快。在一个配置良好的系统中，垃圾回收应当只占用整体执行时间的一小部分，如果更快的回收器会给赋值器操作带来更多的额外开销，则很有可能导致应用程序的整体执行时间变长。赋值器的开销可以是显式的，例如引用计数算法中的读写屏障，但某些隐式因素也可能影响赋值器的性能，例如，如果复制式回收器重排列对象的方式不恰当，则可能降低赋值器的高速缓存友好性；再如，减少引用计数的操作很可能需要访问一个较“冷”的对象。在任何情况下，避免进行同步操作都十分重要，但引用计数的变更必须使用同步操作以避免更新操作的“丢失”，延迟引用计数和合并引用计数则可以消除这些同步操作的开销。

77

有人会使用算法复杂度来比较不同的回收算法。标记-清扫回收需要考虑追踪（标记）和清扫两个阶段的开销，而复制式回收器的复杂度则仅取决于追踪阶段。追踪过程只需要访问所有存活对象，而清扫过程则需要访问每个对象（包括存活对象以及死亡对象）。如果仅据此进行比较，很容易得出标记-清扫回收的开销大于复制式回收这一错误结论。同样是进行追踪，标记-清扫算法访问一个对象所需的指令远比复制式回收算法要少。局部性也对回收性能有很大影响。我们在 2.6 节看到，使用预取技术可以弥补高速缓存不命中问题，但对于复制式回收器而言，是否可以在保留深度优先复制所带来的好处的前提下使用预取技术，

却没有一个完美的答案。在所有的追踪式回收器中,指针追踪的开销通常起决定性作用。另外,当堆中存活对象的比例较小时,复制式回收器表现最佳,但如果在标记-清扫算法中使用懒惰清扫,也可以在这一场景下达到最佳性能。

6.2 停顿时间

许多用户关注的另一个问题是,垃圾回收会给程序的执行造成停顿。尽量缩短停顿时间不仅对交互式程序十分重要,而且是事务型服务处理程序的一个关键要求,否则将导致事务的积压。目前为止我们介绍过的追踪式回收器都会引入万物静止式的停顿,即回收器需要将赋值器线程挂起直到回收完成。在一些较早的系统中,垃圾回收所造成的停顿时间相当惊人,即使在现代硬件条件下,如果使用万物静止式回收,大多数程序的停顿时间也经常会超过 1s。引用计数算法的优势在于它可以将回收开销分摊在程序的执行过程中,从而避免万物静止式的停顿,但正如上一章所提到的,在高性能引用计数系统中,这一优势也并非绝对:当删除指向较大数据结构的最后一个引用时,可能会引发递归性的引用计数修改以及对对象释放操作。所幸的是,对垃圾对象进行引用计数变更操作并不会存在多线程竞争问题,尽管这仍有可能造成对象所在高速缓存行的冲突。更致命的问题是,延迟引用计数和合并引用计数这两种最能有效提升引用计数性能的策略都需要通过万物静止式的停顿来回收零引用表中的对象。我们将在 6.6 节看到,尽管高性能引用计数与追踪式回收在表面上存在很大差异,但它们在本质上并无较大区别。

6.3 内存空间

如果物理内存较小,或者应用程序非常庞大,又或者应用程序需要较好的扩展能力,则内存的使用量便显得十分重要。所有的垃圾回收算法都会引入空间上的开销,这通常是由多方面因素决定的。某些算法可能需要在每个对象上占用一定的空间,例如引用计数域。半区复制式回收器需要额外的堆空间来作为复制保留区,且为了确保回收的安全性,复制保留区应当能够容纳当前所有已分配的对象,除非存在后备的处理机制(例如标记-整理算法)。非移动式回收器会面临内存碎片问题,这将导致堆的可用率降低。回收所需的元数据空间虽然不属于堆,但是也不能忽略。追踪式回收器可能会需要标记栈、标记位图或者其他更加复杂的数据结构。包括显式管理器在内的所有非整理式回收器都需要一定的空间来维持其自身所需的数据结构,例如分区空闲链表等。最后,对于追踪式回收或者延迟引用计数回收而言,如果要避免因频繁回收而导致的性能颠簸,则必须在堆中为垃圾对象保留一定的空间。在垃圾回收系统中,保留空间的大小通常会达到应用程序所需最小内存量的 30% ~ 200%,甚至是 300%。许多系统在必要时可以进行堆扩展,以达到避免性能颠簸等目的。Herts 和 Berger[2005] 指出,使用垃圾回收器的应用程序时如果要达到与显式管理堆相同的性能,其所需的内存空间通常是后者的 3 ~ 6 倍。

78

当某一对象与存活对象图不再关联时,简单的引用计数算法可以立即将其回收。除了可以避免堆中垃圾的堆积,这一特性还具有其他一些潜在优势:被释放的空间通常会在很短时间内重新得到分配,从而有助于高速缓存性能的提升;在某些场景下,编译器能够探测出某一对象成为垃圾的时刻,然后可以立即将其复用,从而无需再将其交给内存管理器去回收。

理想的垃圾回收器不仅应当满足完整性要求(即所有死亡对象最终都会被回收),还应

当达到回收的及时性（即在每个回收周期内都可以将所有死亡对象回收）。前面几章介绍的基本的追踪式回收器都可以达到这一要求，但其代价是每次回收过程都需要扫描所有存活对象。然而，基于性能方面的考虑，现代高性能回收器通常都会放弃回收的及时性，即允许部分垃圾从当前回收周期“漂浮”到下一个回收周期。此外，引用计数算法还面临着回收完整性问题，即如果不借助于追踪方法，环状垃圾便无法得到回收。

6.4 回收器的实现

正确地实现垃圾回收算法并非易事，而正确地实现并发回收算法则更是难上加难。回收器和编译器之间的接口十分关键。回收器所产生的错误很可能在很久之后才会表现出来（或许在多个回收周期之后），而其后果则通常是赋值器尝试去访问一个非法引用，因此回收器的鲁棒性（robustness）与其速度同样重要。Blackburn 等 [2004a] 指出，回收器是关乎性能的关键系统组件，可借助于模块化和组件化等优秀软件工程实践来指导回收器的实现，从而确保代码具有较高的可维护性。

简单的追踪式回收器的优点之一是回收器和赋值器之间的接口比较简单，即只有当分配器耗尽内存时才会唤起回收器。实现这一接口的主要复杂点在于如何判断回收的根，包括全局变量、寄存器和栈槽中包含的引用等，第 11 章将详细讨论这一点。需要强调的是，复制式与整理式回收器的设计要比非移动式回收器复杂得多。移动式回收器需要精确地找出每一个根并更新指向某一对象的所有引用，而非移动式回收器则只需要找到指向存活对象的至少一个引用，也不需要改变指针的值。所谓的保守式回收器（conservative collector）[Boehm and Weiser, 1988] 可以在没有精确的赋值器栈以及对对象布局信息的条件下进行垃圾回收，它们使用智能（但安全、保守）的猜测来确定一个值是否为指针。由于非移动式回收器不会更新引用，因此即使回收器错误地将某个值识别为引用，也不会对该值进行修改，唯一的风险在于可能导致空间的泄漏。关于保守式垃圾回收器的更详细讨论可以参考 Jones [1996, 第 9 章、第 10 章]。

引用计数算法需要与赋值器紧耦合，这既是其优点也是其缺点。其优点在于，引用计数能够以库的形式实现，因而开发者可以自行决定哪些对象需要由引用计数进行管理，而哪些对象需要手动管理。而缺点在于，这种耦合关系给赋值器带来了处理上的开销，而为了确保引用计数操作的正确性，这些操作又至关重要。

对于任何一个使用动态内存分配的现代编程语言，内存管理器都对性能有着至关重要的影响。其关键操作通常包括内存分配、赋值器更新操作（包括读写屏障）、垃圾回收器的内部循环等。这些关键操作的实现代码应当内联（inline），但同时也要小心地避免代码过度膨胀。如果处理器的指令高速缓存足够大，且代码膨胀得足够小（对于高速缓存较小的老式系统，Steenkiste [1989] 建议小于 30%），则代码膨胀对性能的影响可以忽略不计。多数情况下所执行的代码序列（即“快速路径”）应当尽量短小以便于内联，而对于少数情况下才执行的“慢速路径”，则可以用过程调用的方式实现 [Blackburn and McKinley, 2002]。另外，编译器的输出结果也至关重要，因此有必要对其汇编代码进行检查。高速缓存相关行为对性能也有重要影响。

6.5 自适应系统

商业系统通常允许用户自主选择垃圾回收器，且每种回收器都具有一系列可调参数，但

如何进行选择和调整则通常让用户困惑。更加复杂的问题在于,每种回收器的各个参数之间并非相互独立。一些研究人员建议,系统应当能够根据所服务的应用程序所在的环境进行自适应调整。Soman 等 [2004] 所开发的 Java 运行时系统能够在运行时根据可用堆的大小动态地切换回收器的类型,切换的依据有二:一是使用离线分析的方法确定在程序当前堆大小情况下最佳的分配器;二是根据程序当前所用空间占最大可用空间的比例来切换回收器。Singer 等 [2007a] 使用机器学习技术对程序的某些静态特征进行分析,并据此预测最适用于该程序的回收器类型(仅需要一次实验运行)。Sun 的 Ergonomic 调节系统^①可以通过对 HotSpot 回收器性能的调整来控制堆中可用空间的大小,进而达到用户所需的吞吐量以及最大停顿时间要求。

对于开发者而言,我们能够提供的最好的或许也是唯一的建议是,掌控自己所开发应用程序的行为以及所用对象的空间大小、生命周期分布特征,然后据此使用不同的回收器进行实验,并最终选用最合适的一种。但是,实验通常需要基于真实的数据集才能保证结果的准确性,人造的、“玩具”般的基准测试程序都可能会起反向误导作用。

6.6 统一垃圾回收理论

前面的章节介绍了两种不同类型的垃圾回收策略:一种是直接回收,即引用计数;另一种是间接回收,即追踪式回收。Bacon 等 [2004] 发现这两种回收策略之间存在深层次的相似性,并提出了统一垃圾回收理论这一抽象框架。我们可以据此精确地展示不同回收器之间的相同与差异。

80

6.6.1 垃圾回收的抽象

在接下来的抽象框架中,我们仅使用简单的抽象数据结构,具体的实现方式可以有所不同。垃圾回收可以表示为一种定点计算(fixed-point computation),即计算某一节点 n 的引用计数 $\rho(n)$ 。对象的有效引用来源包括根集合以及其他引用计数非零的节点,即:

$$\begin{aligned} \forall \text{ref} \in \text{Nodes}: \\ \rho(\text{ref}) = & |\{\text{fld} \in \text{Roots} : * \text{fld} = \text{ref}\}| \\ & + |\{\text{fld} \in \text{Pointers}(n) : n \in \text{Nodes} \wedge \rho(n) > 0 \wedge * \text{fld} = \text{ref}\}| \end{aligned} \quad (6.1)$$

从引用计数角度考虑,引用计数非零的对象应当保留,而剩余的对象则应被回收。引用计数的值不必十分精确,但至少应当是其真实值的一个安全近似。在抽象的垃圾回收算法中,对引用计数的计算需要用到一个待处理对象工作列表 W ,当 W 为空时算法结束。在接下来的描述中, W 是一个多集合(multiset),因为同一个引用可能会在不同的操作中多次被添加到其中。

6.6.2 追踪式垃圾回收

统一垃圾回收理论将追踪式垃圾回收算法抽象成一种引用计数形式。算法 6.1 展示了追踪式垃圾回收过程的抽象:追踪过程从所有引用计数非零的节点出发;每个回收周期结束后,sweepTracing 方法会将所有节点的引用计数清零,New 方法也将新创建对象的引用计数置为零;在 collectTracing 方法中,算法先调用 rootsTracing 方法构建初始的工作列表 W ,然后将其传递给 scanTracing 方法。

① <http://java.sun.com/docs/hotspot/gc5.0/ergo5.html>。

算法 6.1 追踪式垃圾回收抽象

```

1  atomic collectTracing():
2      rootsTracing(W)
3      scanTracing(W)
4      sweepTracing()
5
6  scanTracing(W):
7      while not isEmpty(W)
8          src ← remove(W)
9           $\rho(\text{src}) \leftarrow \rho(\text{src}) + 1$                                 /* src 从白色变成黑色 */
10         if  $\rho(\text{src}) = 1$                                            /* node 为白色 */
11             for each fld in Pointers(src)
12                 ref ← *fld
13                 if ref  $\neq$  null
14                      $W \leftarrow W + [\text{ref}]$ 
15
16  sweepTracing():
17      for each node in Nodes
18          if  $\rho(\text{node}) = 0$                                            /* node 为黑色 */
19              free(node)
20          else                                                         /* 重新将 node 设置为白色 */
21               $\rho(\text{node}) \leftarrow 0$                                 /* ref 为白色 */
22
23  New():
24      ref ← allocate()
25      if ref = null
26          collectTracing()
27          ref ← allocate()
28      if ref = null
29          error "Out of memory"
30       $\rho(\text{ref}) \leftarrow 0$                                            /* node 为黑色 */
31      return ref
32
33  rootsTracing(R):
34      for each fld in Roots
35          ref ← *fld
36          if ref  $\neq$  null
37               $R \leftarrow R + [\text{ref}]$ 

```

正如我们所预期的那样，回收器会对整个对象图进行追踪，以发现所有从根节点可达的对象：scanTracing 方法对工作列表中的每个节点进行追踪，并在追踪过程中将发现的每个节点的引用计数加 1，这样最终便可完成所有节点引用计数的重建（回想我们在 5.6 节所描述过的，追踪式回收器可以用于修正粘性引用计数）。如果某一可达节点 src 首次被发现（即从 0 变为 1 时，见算法 6.1 中的第 10 行），回收器将对其各指针域进行扫描，同时将它们所指向的子节点加入到工作列表 W 中，从而实现对 src 所有出边的递归扫描^①。

while 循环的结束意味着扫描过程的完成，此时所有存活节点都已经被发现，每个存活对象的非零引用计数等于该节点的入边的数量。接下来，sweepTracing 方法释放所有的无用节点，同时将所有对象的引用计数清零，以便进行下一次回收。需要注意的是，实际应用中的追踪式回收器可以仅用一个位来表示对象的引用计数，即通过标记位来记录对象是曾否被访问过，此时的标记位就相当于其引用计数的粗略近似。

① 另外，也可以将对象添加到日志中，虽然这样会占用更多的空间，但是可以提升追踪过程的缓存性能（参见 2.6 节）。此处不采用该方案的原因是它与算法 6.2 中的引用计数抽象具有一定的差别。

追踪式回收器的计算结果是式 (6.1) 的最小定点解 (least fixed-point solution), 即每个对象的引用计数值都是可以满足等式的最小值。

我们可以使用 2.2 节介绍的三色抽象来对垃圾回收算法进行诠释。在算法 6.1 中, 引用计数为零的对象为白色, 而引用计数非零的对象则为黑色。对象颜色从白色到灰色再到黑色的变化过程代表着对象从初次被发现到完成扫描的整个过程。因此, 抽象追踪算法也可以看作是将全部节点划分为两个集合的过程, 黑色对象集合代表可达对象, 白色对象集合代表垃圾。

[81]

6.6.3 引用计数垃圾回收

为展示引用计数垃圾回收与追踪式回收的相似之处, 在算法 6.2 中所展示的抽象引用计数垃圾回收算法中, 由赋值器执行的 `inc` 和 `dec` 方法会将引用计数操作写入缓冲区, 而不是立即执行。对于多线程应用程序而言, 对引用计数变更操作进行缓冲的策略十分有用, 我们将在第 18 章进行更深入的介绍。此处的缓冲操作与 5.4 节所介绍的合并引用计数算法存在相似之处。在算法 6.2 中, 具体的垃圾回收工作是在 `collectCounting` 方法中完成的, 其中 `applyIncrements` 方法执行被延迟的引用计数的增加操作 (I), `scanCounting` 方法执行被延迟的引用计数的减少操作 (D)。

[82]

当赋值器使用 `write` 方法执行赋值操作时, 即将新的目标引用 `dst` 写入域 `src[i]` 中时, 对新目标对象引用计数的增加 (即 `inc(dst)`) 以及对原目标对象引用计数的减少 (即 `dec(src[i])`) 都会写入缓冲区。

在回收开始阶段, 回收器首先执行缓冲区中所有的引用计数增加操作, 此时对象的引用计数可能比其真实值要大。接下来, `scanCounting` 方法将会对工作列表进行遍历, 并将其遇到的每个对象的引用计数减 1^①。如果某个对象的引用计数在这个阶段降为零, 说明该对象是垃圾, 同时其子节点也会被加入到工作列表。最后, `sweepCounting` 方法会释放所有的垃圾节点。

追踪式算法与引用计数算法整体相同, 它们之间仅存在一些细微的差别。两种算法均包含扫描过程: 追踪式回收的 `scanTracing` 方法使用引用计数增加操作, 而引用计数算法的 `scanCounting` 方法则使用引用计数减少操作。两种算法都需要对零引用对象进行递归扫描, 都需要通过清扫过程来释放垃圾节点所占用的空间。算法 6.1 和算法 6.2 中前 31 行的大致框架基本类似。另外, 延迟引用计数策略 (将根对象引用计数操作延迟的策略) 也符合这一抽象框架 (见算法 6.3)。

前面的章节曾经提到, 当对象图中存在环时, 引用计数的计算会遇到问题。图 6.1 展示了一个简单的对象图, 其中的两个对象构成一个独立的环。如果对象 A 的引用计数为零, 则对象 B 的引用计数也为零 (因为只有引用计数非零的对象才会对其所引用对象的引用计数造成影响)。但由于对象 A 和对象 B 的引用计数相互依赖, 因而这里便产生了一个鸡生蛋还是蛋生鸡的问题: 我们也可以将对象 A 的引用计数预设为 1, 从而其所引用的对象 B 的引用计数也为 1。

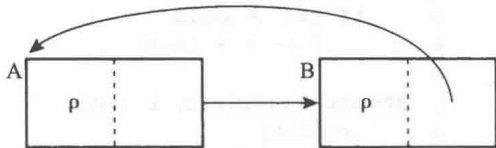


图 6.1 简单的环

由于可能出现多种不同的情况, 所以通用的定点计算表达式可能存在多个不同的解。对于图 6.1 所示的情况, $Nodes = \{A, B\}$, $Roots = \{\}$, 则定点运算 (见式 6.1) 存在两个解:

① 工作列表的初始值即为引用计数减少操作被延迟的对象集合。——译者注

最小定点解 $\rho(A) = \rho(B) = 0$ 和最大定点解 $\rho(A) = \rho(B) = 1$ 。追踪式回收器的计算结果是最小定点解，而引用计数回收器的计算结果则是最大定点解，因而其无法（仅依靠引用计数本身）回收环状垃圾。仅从环状垃圾可达的对象集合是这两个解的唯一不同之处。5.5 节曾介绍过，在引用计数算法中可以利用局部追踪来回收环状垃圾，这一过程本质上就是从最大定点解出发，不断缩小待回收对象的集合，最终得到最小定点解的过程。

算法 6.2 抽象引用计数垃圾回收

```

1  atomic collectCounting(I,D):
2      applyIncrements(I)
3      scanCounting(D)
4      sweepCounting()
5
6  scanCounting(W):
7      while not isEmpty(W)
8          src  $\leftarrow$  remove(W)
9           $\rho(\text{src}) \leftarrow \rho(\text{src}) - 1$ 
10         if  $\rho(\text{src}) = 0$ 
11             for each fld in Pointers(src)
12                 ref  $\leftarrow$  *fld
13                 if ref  $\neq$  null
14                     W  $\leftarrow$  W + [ref]
15
16 sweepCounting():
17     for each node in Nodes
18         if  $\rho(\text{node}) = 0$ 
19             free(node)
20
21
22
23 New():
24     ref  $\leftarrow$  allocate()
25     if ref = null
26         collectCounting(I,D)
27         ref  $\leftarrow$  allocate()
28     if ref = null
29         error "Out of memory"
30      $\rho(\text{ref}) \leftarrow 0$ 
31     return ref
32
33 dec(ref):
34     if ref  $\neq$  null
35         D  $\leftarrow$  D + [ref]
36
37 inc(ref):
38     if ref  $\neq$  null
39         I  $\leftarrow$  I + [ref]
40
41 atomic Write(src, i, dst):
42     inc(dst)
43     dec(src[i])
44     src[i]  $\leftarrow$  dst
45
46 applyIncrements(I):
47     while not isEmpty(I)
48         ref  $\leftarrow$  remove(I)
49          $\rho(\text{ref}) \leftarrow \rho(\text{ref}) + 1$ 

```

算法 6.3 抽象延迟引用计数垃圾回收

```
1  atomic collectDrc(I,D):
2      rootsTracing(I)
3      applyIncrements(I)
4      scanCounting(D)
5      sweepCounting()
6      rootsTracing(D)
7      applyDecrements(D)
8
9  New():
10     ref  $\leftarrow$  allocate()
11     if ref = null
12         collectDrc(I,D)
13         ref  $\leftarrow$  allocate()
14         if ref = null
15             error "Out of memory"
16      $\rho(\text{ref}) \leftarrow 0$ 
17     return ref
18
19 atomic Write(src, i, dst):
20     if src  $\neq$  Roots
21         inc(dst)
22         dec(src[i])
23     src[i]  $\leftarrow$  dst
24
25 applyDecrements(D):
26     while not isEmpty(D)
27         ref  $\leftarrow$  remove(D)
28          $\rho(\text{ref}) \leftarrow \rho(\text{ref}) - 1$ 
```


内存分配

内存管理器需要处理的问题包括3个方面：①如何分配内存；②如何确定存活数据；③如何回收死亡对象所占用的空间，以便在程序的后续执行过程中重新将其分配出去。对于这3个问题，垃圾回收系统和显式内存管理器有着不同的处理策略，且不同回收器所使用的算法也各不相同。但不论如何，内存的分配和回收过程都是紧密相关的，使用任何一种分配策略都必须要考虑如何回收其所分配的内存。

20世纪50年代以来，研究者们针对程序控制下的动态内存分配和释放问题提出了多种解决方案，其中大多数都与垃圾回收系统有着潜在的相关性。然而，自动内存管理和显式内存管理在释放的行为特征方面存在多种显著差别，这决定了它们在分配策略的选择以及性能方面均有所不同。

- 垃圾回收系统一次性完成所有死亡对象的回收，而显式内存管理却通常一次只回收一个对象。更进一步，某些垃圾回收算法（如复制式或整理式回收）可以一次性回收大块连续空间。
- 许多拥有垃圾回收能力的系统在分配对象时可以获取更多的信息，例如待分配对象的大小及其类型。
- 相对于显式内存管理，垃圾回收可以将开发者从内存管理的任务中解脱出来，从而其编程模式会更倾向于频繁地使用堆内存分配。

我们将采用与 Standish[1980] 相同的分类方法来描述主要的几种分配策略，然后再进一步讨论上述几个方面将如何影响垃圾回收系统的分配器设计。

本章将首先介绍两种基本的分配策略：顺序分配（sequential allocation）与空闲链表分配（free-list allocation），紧接着是基于多空闲链表（multiple free-list）的更加复杂的分配。然后将介绍其他一些实用的注意事项。本章最后将对选择分配策略时需要考虑的因素进行总结。

7.1 顺序分配

顺序分配使用一个较大的空闲内存块。对于 n 字节的内存分配请求，顺序分配将从空闲块的一端开始进行分配，其所需的数据结构十分简单，只需要一个空闲指针（free pointer）和一个界限指针（limit pointer）。算法 7.1 展示了顺序分配的伪代码，其内存分配方向是从低地址到高地址，图 7.1 对其分配过程进行了描述。由于顺序分配策略总是简单地移动空闲指针，所以俗称为阶跃指针分配（bump pointer allocation）。在某些场景下，顺序分配也被称为线性分配（linear allocation），因为对于指定内存块而言，其所分配地址的顺序总是线性的。关于分配过程中的字节对齐以及填充（padding）要求，参见 7.6 节和 7.8 节。顺序分配的特征如下：

- 简单。
- 高效（尽管 Blackburn 等 [2004] 指出，对于 Java 系统而言，顺序分配和分区适应空闲链表分配（见 7.4 节）之间的性能差距不超过 1% 的整体执行时间）。

- 相对于空闲链表分配，顺序分配可以给赋值器带来更好的高速缓存局部性，特别是对于移动式回收器中对象的初次分配。
- 与空闲链表分配相比，顺序分配不适用于非移动式回收器。如果未被回收的对象将较大的空闲内存块分割成许多较小的内存块，则空闲内存将会呈现出碎片化趋势，即可用空间散布在众多可以用于顺序分配的小内存块中，而不是少数几个大内存块里。

算法 7.1 顺序分配

```
1 sequentialAllocate(n):
2   result ← free
3   newFree ← result + n
4   if newFree > limit
5     return null
6   free ← newFree
7   return result
```

/* 内存耗尽 */

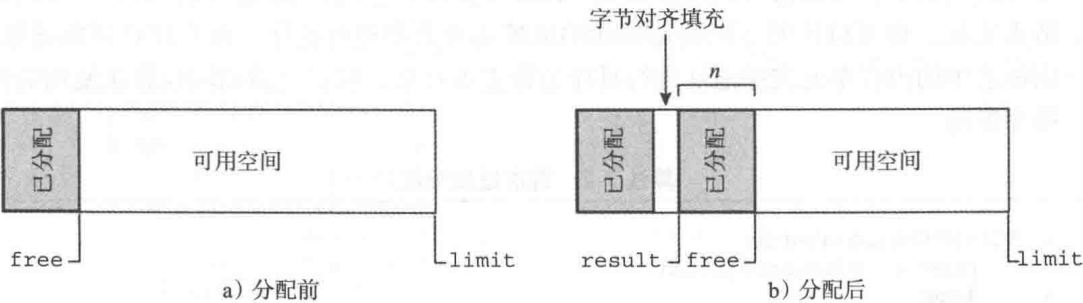


图 7.1 顺序分配：调用 sequentialAllocate(n)，空闲指针将增加 n 个字节，即所需要的内存大小。如果存在字节对齐要求，则可能要额外增加数个填充字节

7.2 空闲链表分配

空闲链表分配是与顺序分配截然不同的一种内存分配策略，它使用某种数据结构来记录空闲内存单元（free cell）的位置和大小，该数据结构即空闲内存单元的集合。严格意义上讲，空闲内存单元的组织方式并非一定是链表，也可以采用其他形式，但尽管如此，我们仍使用“空闲链表”这一传统名称。我们可以将顺序分配看作是空闲链表分配退化后的一种特例，但在实际应用中，顺序分配的性质更加特殊，且实现更为简单。

88

我们将首先考虑以单链表方式组织空闲内存单元的策略，即在需要内存分配时，分配器顺序检测每个空闲内存单元，依照某种策略选择一个并从中进行分配。其算法实现通常是顺序扫描所有空闲内存单元，直到发现第一个符合条件的内存单元为止，因而被称为顺序适应分配（sequential-fits allocation）。典型的顺序适应策略包括首次适应（first-fit）、循环首次适应[⊖]（next-fit）、最佳适应（best-fit）[Knuth, 1973, 2.5 节]，我们将逐一进行描述。

7.2.1 首次适应分配

对于一次内存分配请求，首次适应分配器将在所发现的第一个满足分配要求的内存单元中进行分配。如果该内存单元的空间大于所需的分配空间，则分配器将分裂（split）该内存单元，并将剩余空间归还到空闲链表中。然而，如果分裂后的剩余空间太小以至于无法再用

⊖ 国内通用说法。——译者注

89

于分配（分配算法及数据结构通常限制了最小可分配内存单元的大小），则分配器会避免分裂。另外，如果分裂后的剩余空间小于某一固定阈值或者某一百分比，分配器也可避免分裂。算法 7.2 给出了首次适应分配的代码。需要注意的是，该算法假定每个空闲内存单元内部都记录了自身大小以及下一个空闲内存单元的地址，因此只需要一个全局变量 head 来记录链表中第一个空闲内存单元的地址。

算法 7.3 是首次适应分配的一个变种，即当节点分裂之后将后半段内存单元分配出去，从而简化了代码实现。该方案可能的不足之处在于其对象对齐方式有所不同，但无论如何这也是分割内存单元的一种方式。首次适应分配的特征如下 [Wilson 等, 1995a, 第 31 页]：

- 较小的空闲内存单元会在靠近链表头的位置聚集，从而导致分配速度变慢。
- 在空间使用率方面，在首次适应分配算法的空闲链表中，内存单元会大致以从小到大的顺序排列，因此其行为模式与最佳适应分配比较相似。

在首次适应分配算法中，空闲节点在链表中的排列顺序是一个值得关注的问题。在显式内存管理的场景下，分配器可以将被释放的内存单元插入空闲链表的不同位置，例如链表头部、链表末尾，也可以按照空闲内存单元的地址或者大小进行排序。而在垃圾回收场景中，对空闲链表中的内存单元按照地址进行排序通常更加自然，标记-清扫回收算法使用的便是这一排序策略。

算法 7.2 首次适应分配

```

1 firstFitAllocate(n):
2     prev ← addressOf(head)
3     loop
4         curr ← next(prev)
5         if curr = null
6             return null                /* 内存耗尽 */
7         else if size(curr) < n
8             prev ← curr
9         else
10            return listAllocate(prev, curr, n)
11
12 listAllocate(prev, curr, n):
13     result ← curr
14     if shouldSplit(size(curr), n)
15         remainder ← result + n
16         next(remainder) ← next(curr)
17         size(remainder) ← size(curr) - n
18         next(prev) ← remainder
19     else
20         next(prev) ← next(curr)
21     return result

```

算法 7.3 首次适应分配：空闲内存单元分裂的另一种策略

```

1 listAllocateAlt(prev, curr, n):
2     if shouldSplit(size(curr), n)
3         size(curr) ← size(curr) - n;
4         result ← curr + size(curr)
5     else
6         next(prev) ← next(curr)
7         result ← curr
8     return result

```

7.2.2 循环首次适应分配

循环首次适应分配 (next-fit allocation) 是首次适应分配算法的一个变种。在分配内存时, 该算法不是每次都从空闲链表头部开始查找, 而是从上次成功分配的位置开始 [Knuth, 1973], 即算法 7.4 中的 `prev` 变量。当查找过程到达链表末尾时, 指针 `curr` 将绕回到表头继续进行查找。与首次适应分配相比, 该算法可以有效避免对空闲链表前端较小空闲内存单元的遍历。虽然循环首次适应分配看起来很有吸引力, 但在实践中它通常表现较差:

- 在空间上相邻的存活对象可能并不是在同一时段分配, 因此它们被回收 (或者显式释放) 的时间也通常不同, 从而加剧了内存碎片化 (见 7.3 节)。
- 在分配与释放的过程中, 空闲指针的位置会不断向前迭代, 导致新分配出去的空间并不是刚刚被释放的内存单元, 因此其空间局部性较差。
- 在同一时段内分配的对象会散落在堆中不连续的位置上, 且穿插在其他时段分配的对象之间, 从而降低了赋值器的局部性。

算法 7.4 循环首次适应分配

```

1 nextFitAllocate(n):
2     start ← prev
3     loop
4         curr ← next(prev)
5         if curr = null
6             prev ← addressOf(head)      /* 绕回空闲链表的头部重新开始 */
7             curr ← next(prev)
8         if prev = start
9             return null                  /* 内存耗尽 */
10        else if size(curr) < n
11            prev ← curr
12        else
13            return listAllocate(prev, curr, n)

```

7.2.3 最佳适应分配

所谓最佳适应分配是指在空闲链表中找到满足分配要求且空间最小的空闲内存单元, 其目的在于减少空间浪费, 同时避免不必要的内存单元分裂。算法 7.5 演示了最佳适应分配的具体实现。在实际应用中, 最佳使用分配似乎在大多数应用程序中都表现良好, 其空间浪费率相对较低, 但在最差情况下性能较差 [Robson, 1977]。尽管这一测试结果是在显式内存释放实验中得出的, 但可以预测, 其内存浪费率较低的特征在垃圾回收系统中仍然成立。

90

算法 7.5 最佳适应分配

```

1 bestFitAllocate(n):
2     best ← null
3     bestSize ← ∞
4     prev ← addressOf(head)
5     loop
6         curr ← next(prev)
7         if curr = null || size(curr) = n
8             if curr ≠ null
9                 bestPrev ← prev
10                best ← curr
11            else if best = null

```

```

12         return null /* 内存耗尽 */
13     return listAllocate(bestPrev, best, n)
14 else if size(curr) < n || bestSize < size(curr)
15     prev ← curr
16 else
17     best ← curr
18     bestPrev ← prev
19     bestSize ← size(curr)

```

7.2.4 空闲链表分配的加速

如果堆空间较大，则仅使用单个顺序链表对空闲内存单元进行组织便显得力不从心，因此研究者们开发出了多种更加复杂的空闲内存单元组织方式来加快它的分配速度。一种显而易见的方案是使用平衡二叉树来组织空闲内存单元，从而可以按照空间大小（针对最佳适应分配）或者地址顺序（针对首次适应或者循环首次适应分配）进行排序。当按照节点大小进行排序时，可以将大小相同的空闲节点组织成一条链表，然后使用平衡二叉树对各条链表进行管理。此时不仅查询效率较高，而且由于节点的分配和归还通常都在对应的链表中完成，因而树的变更操作较少，整体分配效率可以得到提升。

笛卡儿树 [Vuillemin, 1980] 是适用于首次适应分配或者循环首次适应分配的平衡树，它同时使用节点的地址（主键）和大小（次键）来组织索引。笛卡儿树依照节点地址进行排序，同时也将节点按照空间大小组织成“堆”，从而允许在首次适应或循环首次适应分配中快速找到满足要求的节点。这一策略同时也被称为快速适应分配（fast-fit allocation）[Tadman, 1978；Standish, 1980；Stephenson, 1983]。对于笛卡儿树中的某一节点，其所记录的内容包括：该节点所对应空闲内存单元的地址和大小、左右子节点的指针、节点自身及其子树中最大空闲内存单元的大小（该值的计算方法是先获取节点自身所对应空闲内存大小、左右子节点的最大空闲内存大小，然后取三者的最大值），因此笛卡儿树的节点大小会比基于链表的简单方案要大。算法 7.6 展示了基于笛卡儿树实现首次适应分配的查找过程，其中忽略了树中节点的插入和删除操作。全局变量 root 代表笛卡儿树的根节点，max(n) 表示节点 n 的子树（包括节点 n 自身）中最大空闲内存单元的大小。循环首次适应分配的代码仅比首次适应分配稍为复杂一些。

算法 7.6 使用笛卡儿树查找空闲节点

```

1 firstFitAllocateCartesian(n):
2     parent ← null
3     curr ← root
4     loop
5         if left(curr) ≠ null && max(left(curr)) ≥ n
6             parent ← curr
7             curr ← left(curr)
8         else if prev < curr && size(curr) ≥ n
9             prev ← curr
10            return treeAllocate(curr, parent, n)
11        else if right(curr) ≠ null && max(right(curr)) ≥ n
12            parent ← curr
13            curr ← right(curr)
14        else
15            return null /* 内存耗尽 */

```


平衡二叉树的引入将分配算法在最坏情况下的时间复杂度从 $O(N)$ 降低到 $O(\log(N))$ ，其中 N 为空闲内存单元的数量。自调整树 (self-adjusting tree) (splay tree) [Sleator and Tarjan, 1985] 也具有类似的优点 (即平均分配速度较快)。

对于首次适应或循环首次适应分配而言，将空闲内存单元依照地址进行排序的另一种有效策略是位图适应分配 (bitmapped-fits allocation)。该算法使用额外的位图来记录堆中每个可分配内存颗粒的状态，因此在进行内存分配时，分配器可以基于位图而非堆本身进行搜索。借助一张预先计算好的映射表，分配器仅需要对位图中一个字节进行计算，便可得知其所对应的 8 个连续内存颗粒所能组成的最长连续可用空间。也可以使用额外的长度信息记录较大的空闲内存单元或者已分配内存单元，从而快速将其跳过以提升分配性能。位图适应分配具有如下一些优点：

- 位图本身与对象相互隔离，因此不容易遭到破坏。这一特性不仅对于诸如 C 和 C++ 等安全性稍低的语言十分重要，而且对于安全性更高的语言也十分有用，因为这可以提升回收器的可靠性以及可调试性。
- 引入位图之后，无论是对于空闲内存单元还是已分配内存单元，回收器都不需要占据其中的任何空间来记录回收相关信息，从而最大限度地降低了对内存单元大小的要求。如果以一个 32 位的字作为最小内存分配单元，该策略会引入大约 3% 的空间开销，但其所带来收益却远大于这一开销。不过，基于其他一些方面的考虑，对象可能依然需要一个头部，因而这一优点并非始终能够得到体现。
- 相对于堆中的内存单元，位图更加紧凑，因此基于位图进行扫描可以提升高速缓存命中率，从而提升分配器的局部性。

7.3 内存碎片化

对于支持动态内存分配的系统，在程序执行的初始阶段，堆中通常仅包含一个或者少数几个大块连续空闲内存。随着程序执行过程中不断的内存分配与释放，堆中逐渐会出现许多较小的空闲内存单元。我们将这种大块可用内存空间被拆分成大量小块可用内存的现象称为内存碎片化 (fragmentation)。对于动态内存分配系统而言，碎片化至少带来两种负面影响：

- 导致内存分配失败。对于一次内存分配请求，尽管堆的整体空闲内存足够，但可能所有的空闲内存单元都无法满足分配需求。对于非垃圾回收系统，这一情况通常会导致程序崩溃。而对于垃圾回收系统而言，这可能加快垃圾回收的频率。
- 即使堆中的空闲内存可以满足分配需求，碎片化问题仍可能导致程序消耗更多的地址空间、更多的常驻内存页以及更多的高速缓存行。

想要完全避免内存碎片是不切实际的。一方面，分配器通常无法预测程序未来会以何种序列进行内存分配；另一方面，即使可以精确地预测内存分配序列，找出一种最优的内存分配策略 (即使用最小的空间来满足一组内存分配与释放序列) 也是 NP 困难的 [Robson, 1980]。尽管我们并不能根除内存碎片化，但仍可以找到一些较好的方法来对其进行控制。一般来说，我们应当在分配速度和碎片化之间进行一个粗略的平衡，同时我们也发现，在任何情况下对内存碎片化进行预测都是十分困难的。

我们可能会在直觉上认为最佳适应分配的内存碎片较少，但它却会导致堆中散布大量很小的内存碎片。首次适应分配也会产生大量小块内存碎片，它们通常集中在靠近空闲链表头的位置。循环首次适应分配趋向于将小块碎片均匀地分散在堆中，但并不是说这样就更好。唯一可以解决内存碎片化问题的方案是使用整理式或者复制式垃圾回收。

7.4 分区适应分配

93

基本的空闲链表分配器会将大多数时间花费在合适的空闲内存单元的查找上，因此如果使用多个空闲链表对不同大小的空闲内存单元进行管理，则会加快分配速度 [Comfort, 1964]。在本节所述的“分区适应”概念中，多个空闲链表所服务的仅仅是相同的逻辑空间，而在第9章我们将看到，回收器也可以将堆划分为“多个内存空间”，且每个内存空间拥有其专属的内存分配器，我们必须对这两个概念进行区分。例如，许多回收器会对大对象或者不包含任何引用的大对象（例如图像或者其他二进制数据）进行单独管理，这不仅是基于性能的考虑，而且是因为这些对象通常都具有与众不同的生命周期特征。这些大对象可能会位于不同的空间，且回收器会对它们进行特殊处理。另外，每个内存空间内部都可能存在独立的大对象集合，且它们通常使用分区适应算法进行分配，但是内存空间中的小对象却通常使用顺序分配而非分区适应分配。有许多方法可以将分区适应分配与多内存空间策略相结合。

分区适应分配的基本思想是将可分配内存单元的大小限制为 k 种，其中， $s_0 < s_1 < \dots < s_{k-1}$ 。 k 值的选择可以有多种，但其通常是一个固定值。空闲链表通常有 $k+1$ 个，即 f_0, \dots, f_k 。在空闲链表 f_i 中，空闲内存单元的大小 b 必须满足 $s_{i-1} < b \leq s_i$ ，其中， $s_{-1} = 0$ ， $s_k = +\infty$ 。由于分区的目的在于避免内存分配时的空闲内存单元查找过程，所以我们可以进一步将空闲链表 f_i 中内存单元的大小精确地限制为 s_i ，但空闲链表 f_k 是一个例外，它将用于保存所有大于 s_{k-1} 的空闲内存单元。当需要分配不大于 s_{k-1} 的内存单元时，分配器会将所需的空间大小 b 向上圆整到不小于 b 的最小的 s_i 。我们将不同大小的 s_i 称为空间大小分级（size class），因此针对 b 的空间大小分级即为满足 $s_{i-1} < b \leq s_i$ 的一个。

空闲链表 f_k 中所管理的是所有大于 s_k 的空闲内存单元，我们可以使用 7.2 节所描述的某种单链表算法进行管理，因此笛卡儿树或者其他具有较好的期望时间性能的数据结构都是不错的备选方案。一方面，大对象的分配通常不太频繁；另一方面，即使抛开分配频率这一因素，仅对较大对象进行初始化就会付出较大开销。因此，即使大对象的分配开销比在单一空间大小的链表中进行分配稍高，其对程序整体执行时间的影响也不会超过 1%。

对于大小为 b 的内存分配需求，有多种方式可以加速对应的空间大小分级的计算。假设 s_0 到 s_{k-1} 之间的空间大小等级均匀分布，即 $s_i = s_0 + c \cdot i$ ，且 $c > 0$ ，如果 $b > s_{k-1}$ ，则对应的空间大小分级为 s_k ，否则将为 s_j ，其中 $j = [(b - s_0 + c - 1) / c]$ （即线性适配，表达式中加上 $c - 1$ 的目的是为了向上圆整）。例如，假定某一分配策略的参数是： $s_0 = 8$ ， $c = 8$ ， $k = 16$ ，这意味着 $8 \sim 128$ 中每个能够被 8 整除的数字都是一个空间大小分级，且 $b > 128$ 的内存分配需求将交由通用的空闲链表算法进行管理。对于以字节（byte）为单位进行寻址的计算机，分配的单位通常为字节；而对于以字（word）为单位进行寻址的计算机，分配的单位通常是字。即使分配单位是字节，内存颗粒的大小通常也会是一个字或者更大。如果 c 是 2 的整数次幂，则表达式中的除法操作可以用移位的方式实现，这将比通用的除法操作快得多。

较小的空间大小分级可以分布得较为密集。除此之外，为避免对通用空闲链表分配算法的调用，分配器也可以提供多个较大的、分布较为稀疏的空间大小分级。例如，在 Boehm-Demers-Weiser 回收器中，不大于 8 字节的对象均会在 8 字节的空闲链表中分配，然后在 $16 \sim 32$ 字节中，每个能被 4 整除的数字都会对应一级空闲链表 [Boehm and Weiser, 1988]。对于大于 32 字节的分配需求，则需动态确定其对应的空闲链表，即：使用一个数组将所需内存大小（以字节为单位）映射到对应的分配大小（以字为单位），然后在空闲链表数组中直

接以分配大小为索引找到对应的空闲链表，这些空闲链表均使用懒惰填充策略。

如果将空间大小分级的集合固化在运行时系统中（即在系统编译时就已经确定），则理论上对于任何在编译期可以确定大小的分配需求，编译器都可以预先确定其所对应的空闲链表，这通常可以提高大多数分配操作的性能。

94

基于单个空闲链表的顺序分配（如首次适应、最佳适应等算法）通常会花费较长的时间以寻找合适的空闲内存单元，而如果引入某种形式的平衡树，则在最差情况下的时间复杂度会降低到对数级别。相比之下，分区适应分配的主要优势在于，任何在非 s_k 的空间大小分级中执行的分配都会在常数时间内完成，如算法 7.7 所示，也可参见算法 2.5 中的懒惰清扫变体。

算法 7.7 分区适应分配

```

1 segregatedFitAllocate(j):           /* j 为空间大小分级  $s_j$  的索引号 */
2     result ← remove(freeLists[j])
3     if result = null
4         large ← allocateBlock()
5         if large = null
6             return null              /* 内存耗尽 */
7         initialise(large, sizes[j])
8         result ← remove(freeLists[j])
9     return result

```

7.4.1 内存碎片

7.2 节所述的简单空闲链表分配器只会面临一种内存碎片情况，即空闲内存单元太小以至于无法满足任何分配需求。由于不可用空间分布在已分配空间之外，因此这种碎片被称为外部碎片（external fragmentation）。而在引入空间大小分级之后，分配器必须将分配需求向上圆整到某一特定的空间大小分级，因此在已分配的内存单元内部就可能存在空间上的浪费，由此造成的碎片被称为内部碎片（internal fragmentation）。分区适应分配需要在内部碎片和空间大小分级数量方面进行平衡。特定的字节对齐要求也会以类似的方式引入碎片，但由于不可用空间位于已分配内存单元之外，所以从严格意义上讲这一情况属于外部碎片。

7.4.2 空间大小分级的填充

在分区适应分配算法中，各级空闲链表的填充也是需要考虑的一个重要方面。我们将介绍两种填充策略，一种策略是单个内存块仅用于填充特定大小的空闲链表，即页簇分配；另一种策略则是基于内存块分裂的策略。

基于内存块的页簇分配（big bag of pages block-based allocation）该方案需要一个块分配器来分配大小为 B 的内存块，且 B 为 2 的整数次幂。对于大于 B 的内存分配需求，将直接从块分配器中获取一组连续内存块。对于 $s < B$ 的空间大小分级，如果分配器需要分配更多这一大小的内存单元，首先需要从块分配器中获取一个内存块，然后再将该内存块分割成大小为 s 的内存单元，并填充到空闲链表中。分配器通常需要记录每个内存块填充了哪个空间大小分级，并将该信息与其他元数据（例如该内存块中内存单元的标记位）一起记录在内存块内部，但 Boehm 和 Weiser[1988] 认为更好的方案是将其记录在独立的空间中，这样一来，如果仅需要对内存块的元数据进行查找和更新，该方案可以减少转译后备缓冲区（translation lookaside buffer）不命中以及缺页异常（page fault）的几率，同时也无需刻意将

每个内存块中的元数据以不同的方式对齐（以避免不同内存块的元数据之间竞争相同的高速缓存集合）。

95

在 2.5 节对懒惰清扫的描述过程中，我们介绍了基本的基于内存块的分配（block-based allocation）策略，该策略的优势在于系统能够以内存块而非单个内存单元为单位来记录元数据，但这样却会使内存碎片问题更加复杂：如果每个内存块仅用于填充一种大小的空闲链表，则每个内存块中（平均）一半的空间会被浪费，而对于特定的空间大小分级，最差情况下内存块的浪费率将达到 $(B - s) / B$ （此时每个内存块中仅有一个内存单元被分配出去）。如果 B 不能被 s 整除，则在内存块的末尾会存在一块小于 s 的空间无法使用^①，对于内存块而言，它属于内部碎片，而对于内存单元而言，它属于外部碎片。

在某些系统中，内存单元所关联的元数据不仅包括其空间大小，还包括该其内所承载的对象的类型。如果单个内存块仅用于分配一种类型的对象，可能会造成较大的内存碎片（因为大小相同但类型不同的两种对象必须从不同的内存块中分配，且由不同的空闲链表进行管理），但对于空间较小且数量较多的类型，如果将其元数据记录在内存块而非内存单元中，则在空间上的节省效果还是相当明显的，例如 Lisp 语言中的 cons 单元。

如果将较小内存单元的元数据与其所在的内存块进行关联，则空闲内存单元的合并将极为简单高效：只有当内存块中所有的内存单元都被释放时，才需要合并空闲内存单元，然后再将整个内存块归还给块分配器。对于一般的内存分配需求，分配器只需简单地从对应的空闲链表中分配一个内存单元，如果空闲链表为空，则直接分配一个内存块并用其填充空闲链表。这一分配过程十分高效，但其主要缺陷在于，最差情况下的内存碎片问题较为严重。

内存块分裂（splitting） 在 7.2 节对几种简单空闲链表分配算法的介绍中，我们已经提到了空闲内存单元的分裂策略，即从较大的空闲内存单元中拆出一块以满足较小的内存分配需求。如果空间大小分级的分布较为密集，则在拆分一个空闲内存单元时，很可能会有一个合适的空闲链表恰好能够接纳剩余的内存单元。如果不希望空间大小分级过密，也可以使用一些特殊的方法来组织空闲链表并达到相同效果。伙伴系统（buddy system）即是满足这一条件的方案之一，其空间大小分级均为 2 的整数次幂 [Knowlton, 1965 ; Peterson and Norman, 1977]。我们可以将一个大小为 2^{i+1} 的空闲内存单元分裂为两个大小为 2^i 的空闲内存单元，同时也可以将两个相邻的大小为 2^i 的空闲内存单元合并成大小为 2^{i+1} 的一个，但进行合并的前提是两个相邻空闲内存单元原本就是由同一个较大的空闲内存单元分裂得到的。在该算法中，大小为 2^i 的空闲内存单元两两成对，因而称之为伙伴。由于伙伴系统的内部碎片通常较为严重（对于任意的内存分配需求，其平均空间浪费率会达到 25%），因此该算法基本已经成为历史，在实践中较少使用。

斐波那契伙伴系统（Fibonacci buddy system）[Hirschberg, 1973 ; Burton, 1976 ; Peterson and Norman, 1977] 是伙伴系统的一个变种，其空间大小分级符合斐波那契序列，即 $s_{i+2} = s_{i+1} + s_i$ ，同时需要选定合适的 s_0 和 s_1 。与传统的伙伴系统相比，该算法相邻空闲内存单元的大小比值更小，因而在一定程度上缓解了内部碎片问题。但该算法的问题在于，在回收完成后将相邻空闲内存单元合并的操作会更加复杂，因为回收器需要判定某一空闲内存单元究竟应当与相邻两个空闲内存单元中的哪一个进行合并。

① 可能是为了减缓内存块前端数据产生缓存冲突的概率，Boehm 和 Weiser[1988] 将这块不可用的内存放置在内存块的前端而非末端。当缓存行较小时其作用较为明显，因为只有当内存单元大于一个缓存行时，这一策略才会起作用。

伙伴系统还存在一些其他变种，具体可以参见 Wise[1978]、Page and Hagins[1986]、Wilson 等 [1995b]。

7.5 分区适应分配与简单空闲链表分配的结合

我们可以将分区适应分配当作单一空闲链表分配的前端加速器，并将回收所得的内存单元置于其空间大小分级所对应的空闲链表中。在进行一次内存分配时，如果发现请求所对应的空闲链表为空，则可以使用最佳适应策略（即沿着空间大小分级增大的方向）在更大的空闲链表中进行查找，当然也可以使用首次适应或循环首次适应策略，其不同之处仅在于发现空闲链表为空之后如何进行处理。不论使用哪种策略，分配器都可以确保查找过程能够在空闲链表 f_k 中结束，该链表中所有的空闲内存单元都比 s_{k-1} 要大，此时便需要使用基于空闲链表的分配策略（首次适应、最佳适应或循环首次适应）来完成分配。

96

该策略的另一种描述方式是：如何在已有的分区适应分配策略上实现对空闲链表 f_k 的管理，其方案通常包含以下几种：

- 将其作为单个空闲链表，从而使用首次适应、最佳适应、循环首次适应或者它们的变种，例如笛卡儿树或者其他可以加速空闲内存单元查找的数据结构。
- 使用基于内存块的分配。
- 使用伙伴系统。

7.6 其他需要考虑的问题

真正的内存分配器通常还要考虑其他一些问题，包括字节对齐（alignment）、空间大小限制（size constraint）、边界标签（boundary tag）、堆可解析性（heap parsability）、局部性（locality）、拓展块保护（wilderness preservation）、跨越映射（crossing map）等，我们将逐一进行讨论。

7.6.1 字节对齐

将对象按照特定的边界要求进行对齐，一方面是底层硬件或者机器指令集的要求，另一方面这样做有助于提升各层次存储器的性能（包括高速缓存、转译后备缓冲区、内存页）。以 Java 语言的 double 数组为例，某些机器可能要求 double 这一双字浮点数必须以双字为边界进行对齐，即其地址必须是 8 的整数倍（地址的后三位为零）。一种简单但稍显浪费的解决方案是将双字作为内存分配的颗粒，即所有已分配或未分配内存单元的大小均为 8 的整数倍，且均按照 8 字节边界对齐。但即便如此，当分配一个 double 类型的数组时，分配器仍需要进行一些额外工作。假设 Java 语言中纯对象（即非数组对象）头部都必须保留两个字，一个指向对象的类型信息（用于虚函数调用、类型判定等），另一个用于记录对象的哈希值以及同步操作所需的锁（这也是一种典型的设计方式）。数组对象则需要第三个字来记录其中元素的个数。如果将这三个头部字保存在已分配内存单元的起始位置，则数组元素就不得不以奇数字为单位进行对齐。如果使用双字作为内存颗粒，则可以简单地用四个字（即两个双字）来保存这三个头部字，然后浪费掉一个。

但如果内存颗粒是一个字，我们则希望尽量减少上述的内存浪费。此时，如果某个空闲内存单元按照奇数字对齐（即其地址模 8 余 4），则我们可以简单地将三个头部字放在内存单元的起始位置，后续的数组元素自然会满足双字的对齐要求。如果某个空闲内存单元按照双

字对齐,则我们必须浪费一个字以满足对齐要求。这一方案增加了分配过程的复杂度,因为某一空闲内存单元是否满足分配需求不仅取决于所需空间的大小,还取决于字节对齐要求,正如算法 7.8 所示。

算法 7.8 结合字节对齐要求进行内存分配

```

1 fits(n, a, m, blk):          /* 判断大小为 blk 的空闲内存是否可以满足分配需求 */
2   /* n 为所需空间大小, a 为空闲内存对齐方式, m 为分配所要求的对齐方式且为 2 的整数次幂 */
3   z ← blk - a                /* 预留 a 的空间 */
4   z ← (z + m - 1) & ~(m - 1) /* 向上圆整到 m 的倍数 */
5   z ← z + a                  /* 补偿 a 的空间 */
6   pad ← z - blk
7   return n + pad ≤ size(curr)

```

7.6.2 空间大小限制

某些回收器要求对象(内存单元)的大小必须大于某一下界。例如,基本的整理式回收要求对象内部至少可以容纳一个指针,还有一些回收器可能需要用两个字来保存锁或状态以及转发指针,这就意味着即使开发者仅需要分配一个字,分配器也必须多分配两个字。如果开发者需要分配不包含任何数据、仅用作唯一标识的对象,原则上编译器无需分配任何空间,但在实际情况下这通常不可行:对象必须要有唯一的地址,因此对象的大小至少应为一个字节。

7.6.3 边界标签

为了确保在释放内存时可以将相邻空闲内存单元合并,许多内存分配系统为每个内存单元增加了额外的头部或者边界标签,它们通常不属于可用内存的范畴 [Knuth, 1973]。边界标签保存了内存单元的大小及其状态(即空闲或已分配),还可以在其中记录上一个内存单元的大小,从而可以快速读取上一个内存单元的状态并判断其是否为空。当内存单元空闲时,边界标签也可用于保存构建空闲链表的指针。基于这些原因,边界标签可能达到两个字或者更大,但如果使用一些额外的方法,并允许在分配和释放的过程中引入一定的额外开销,则仍有可能将边界标签压缩到一个字。

如果使用额外的位图来标记堆中每个内存颗粒的状态,则不仅无需使用边界标签,而且可以增加程序的鲁棒性。这一方法是否会减少空间开销,取决于对象的平均大小以及内存颗粒的大小。

我们进一步注意到,垃圾回收通常会一次性释放大量对象,因此某些特定的算法可能不再需要边界标签,或者其边界标签中需要包含的信息较少。另外,托管语言中对象的大小通常可以通过其类型得出,因而无需使用额外的边界标签来单独记录相关信息。

7.6.4 堆可解析性

在标记-清扫回收的清扫阶段,回收器必须能够顺次逐个遍历堆中的每个内存单元,我们将这一能力称为堆可解析性。尽管对于其他种类的回收器而言,堆可解析性并非不可或缺,但它对于回收器的调试将是十分有用的,因此如果条件允许,支持堆可解析性还是很有必要的。

通常我们只需要支持单方向的堆解析,即沿着地址增大的方向。编程语言通常会在对象

内部使用一到两个字来记录对象的类型以及其他信息，我们称之为对象的头部。例如，在许多 Java 语言的实现中，对象头部通常会占据两个字，一个用于记录对象的类型（指向类型信息的指针，类型信息中会包含该类的方法分派向量，另一个用于记录哈希值、同步信息、垃圾回收标记位等。对于数组而言，如果数组的引用与其首个元素的引用相同，且后继元素在高地址方向顺次连续排列，则通过索引快速获取数组元素这一操作在大多数机器上都可以高效地完成。由于运行时系统以及垃圾回收器通常需要以某种统一的方式来获取对象的类型，所以我们将对象的头部置于它的数据之前，但这样一来，对象的引用便不再是其所占用内存单元的首地址，而是数据区某个域的地址。将对象的头部置于数据之前有助于堆的前向解析。

同样以 Java 系统为例，每个数组实例均需单独记录自身长度。将 `length` 域置于一般对象会用到的两个域之后可以简化堆的解析。此时数组的首个元素将位于内存单元的第三个字，而 `length` 域的索引号是 `-1`，其他两个头域的索引分别为 `-2` 和 `-3`。为确保对象类型的获取方式一致，非数组的纯对象同样也需要将其两个头域置于索引号为 `-2` 和 `-3` 位置，这可能导致索引号为 `-1` 的位置出现空洞，但将对象内部的数据整体前移一个字便可以解决这个问题（此处假定硬件允许依照一个较小的负数常量进行索引，大多数硬件满足这一要求）。另外，即使对象不包含额外的域，对于这一布局策略，依然不会存在任何内存浪费的问题：该对象的引用可以是其后继对象某个头域的地址。图 7.2 对这三种情况都进行了描述。

某些系统会存在使用较小对象覆盖较大对象的情况（例如许多函数式语言会将某个闭包替换为其计算后的值），此时可能出现一些特殊的问题。如果覆盖时不采取额外操作，则回收器在扫描堆的过程中可能会遇到覆盖操作的中间状态，进而引发一些不可预知的错误。Non-Stop Haskell 通过插入一个填充对象（`filler object`）的方式解决了这一问题 [Cheadle 等，2004]。赋值器在创建闭包时通常会预留 1~8 个字的元数据，正常情况下的覆盖操作仅需要在元数据中插入一个适当大小的无指针对象，而很少会出现较大的填充对象，一旦出现则需要动态创建元数据[⊖]。

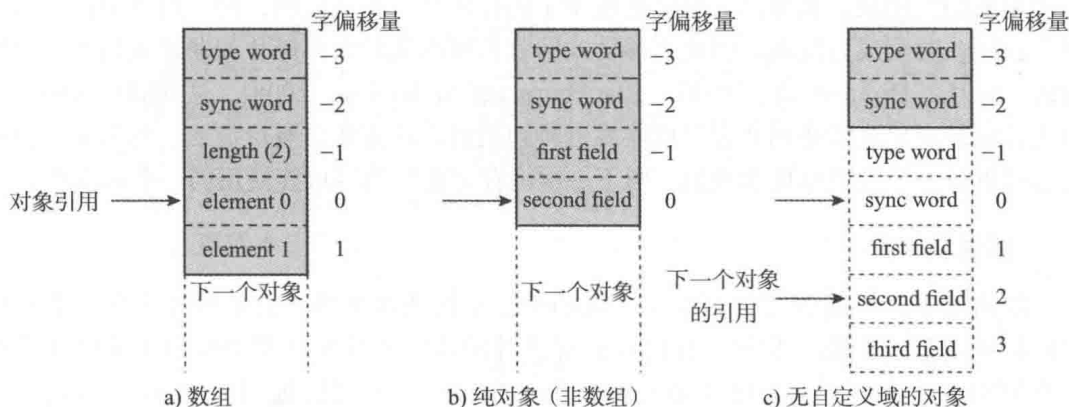


图 7.2 Java 对象头部的内存布局，其目的在于保证堆的可解析性。灰色方框表示对象所占用的空间，虚线框表示相邻对象所占用的空间

最后需要考虑的是字节对齐可能引发的问题。如果出于字节对齐的目的而对某个对象进行一个或者数个字移位，那么我们需要在由此造成的空隙中记录一些数据，以便回收器在

⊖ Cheadle 等人并未提供具体细节，但在这种情况下，我们有理由将元数据置于填充对象中，这样可以避免任何为确保堆可解析性而引入的运行时内存分配。

进行堆解析时将其跳过。如果可以确保对象头部以非零字开始，那么可以将空隙填充为零，并在堆解析时简单地跳过这些为零的字。而另一种简单的方法是将特定范围的值写入空隙的起始位置，该值不仅可以表示接下来的一段内存是空隙，而且可以反映空隙的长度。例如，Sun 公司长期以来都使用一种所谓的“自解析堆”：当（在不可移动的空间中）释放一个对象时，回收器使用一个填充对象覆盖其空间，而填充对象内部会包含一个表示自身大小的域（认为它是一个字数组），清扫器可以据此快速跳过空闲内存单元并找到下一个真正的对象。

如果使用位图来记录每个对象的起始位置，不仅可以简化堆的解析过程，而且降低了对对象头部的设计要求，其不足之处在于位图会占用额外的空间，在分配过程中对位图进行操作也会花费额外的时间。但对于许多回收器而言，基于位图的分配依然十分有用，特别是对于并行回收器与并发回收器。

本节介绍了 Java 语言中一种可以确保堆的可解析性的设计实现，其设计思想同样适用于其他语言。另外，基于内存块的分配不仅会简化小块内存单元的解析，而且也可以简化大块内存的处理。为提升高速缓存性能，我们可以将大对象置于连续内存块中的随机位置^①，这样一来，该对象之前或者之后浪费的空间也是一个随机值。为确保堆的可解析性，可以将大对象的地址简单地记录在内存块的起始位置。

7.6.5 局部性

在内存分配过程中，局部性的影响表现在多个方面。分配和释放过程本身就会受到局部性作用的影响。同等条件下，基于地址顺序的空闲链表分配可能提升分配器访问内存的局部性，顺序分配天然的线性访问模式也具有较高的高速缓存友好性，同时也可以在进行一定的软件预取 [Appel, 1997]，但对于某些硬件而言，软件预取并不是必要的^② [Diwan 等, 1994]。局部性这一概念还会以另一种完全不同的方式影响内存的分配和释放：如果一批对象同时成为垃圾，且它们在堆中集中排列，则当回收完成后，它们所占的空间将会合并成一个单独的空闲块，从而最大限度地减少了内存碎片。事实证明，同一时刻分配的对象通常也会在同一时刻成为垃圾，因此非移动式回收器所面临的内存碎片问题比人们预想的要小 [Hayes, 1991 ; Dimpsey 等, 2000 ; Blackburn and McKinley, 2008]，这同时也说明，将连续两次分配的对象连续排列或者尽可能靠近排列的启发式方法是 valuable 的。如果某次内存分配是通过拆分一个大内存块实现的，则下一次内存分配应当尽可能使用同一个内存块。

7.6.6 拓展块保护

一般情况下，堆通常是由一大块连续的地址空间所组成的，其低地址边界通常与程序的代码段或者静态数据区相邻，高地址边界之外的地址空间则通常会保留下来以备后续扩展。在 UNIX 系统中，这一边界通常被称为“break”，并可以通过 `sbrk` 系统调用进行扩展或者收缩。该边界之外的地址空间通常不会映射到虚拟内存中，因此堆中最后一个空闲内存块便具有了可扩展性，我们称其为“未使用空间”（unoccupied territory），或者拓展块（wilderness）。Korn 和 Vo [1985] 发现，如果将拓展块作为内存分配的最后备选内存块，则有助于降低内存碎片，这一策略被称为拓展块保护。这一策略同时也有助于延缓堆的增长，进而减少整个系统的资源消耗。

① 这一策略可以在一定程度上避免高速缓存冲突。——译者注

② 某些硬件可以探测到用户对地址空间的线性访问模式，进而自动实现硬件预取。——译者注

7.6.7 跨越映射

某些回收策略或者赋值器写屏障需要分配器对跨越映射进行额外的操作。跨越映射反映了堆中每个已对齐的 2^k 片段内最后一个起始于该片段的对象的地址。与堆的可解析性相结合,回收器或赋值器写屏障便可以根据对象内部的某一地址快速找到该对象的起始地址,然后进一步访问该对象头部。11.8 节将讨论跨越映射的更多细节。

7.7 并发系统中的内存分配

在多线程环境下,分配过程的许多操作都需要原子化以确保分配数据结构的完整性,这些操作都必须使用原子操作或者锁,但这样一来,内存分配就可能成为性能瓶颈。最基本的解决方案是为每个线程开辟独立的内存分配空间,如果某个线程的可分配空间耗尽,则从全局内存池中为其分配一个新的空闲块,此时只有与全局内存池的交互才需要原子化。不同线程的内存分配频度可能不同,因此如果在为线程分配内存块时使用自适应算法(即:为分配速度较慢的线程分配较小的内存块,而为分配速度较快的线程分配较大的内存块),则程序的时间和空间性能均可获得提升。Dimpsey 等 [2000] 声称,在多处理器 Java 系统中,为每个线程配备一个合适的本地分配缓冲区(local allocation buffer, LAB)可以大幅提升性能^①。他们进一步指出,由于几乎所有的小对象都是从本地分配缓冲区分配的,因而我们有理由对全局(基于空闲链表的)分配器进行调整,以使其能够更加高效地分配用于线程本地分配缓冲区的内存块。

Garthwaite 等 [2005] 讨论了如何对本地分配缓冲区的大小进行自适应调整,他们同时发现,将本地分配缓冲区与处理器而非线程相关联效果更佳。该算法通过如下方式对本地分配缓冲区的大小进行调整:线程初次申请本地分配缓冲区时将获得 24 个字(94 字节)的内存块,之后每次新申请的内存块均为上一次的 1.5 倍,同时每经历一次垃圾回收过程,回收器都会将线程的本地分配缓冲区的大小折半。该算法同时也会根据不同线程的分配次数调整年轻代的空间大小。每处理器(per-processor)本地分配缓冲区的实现依赖于多处理器的可重启临界区(restartable critical section),Garthwaite 等人对此做了介绍。其基本原理是,线程可以判断自身是否被抢占(preempt)或者被重新调度(reschedule),然后可以据此判断自身是否被切换到其他处理器上运行。当线程抢占发生时,处理器会对某个本地寄存器进行修改,该操作会为抢占完成后的写入操作设置一个陷阱,而陷阱处理函数则会重启被中断的分配过程。尽管每处理器本地分配缓冲区需要更多的指令支持,但与每线程本地分配缓冲区相比,其分配时延相同,且不需要复杂的缓冲区调整机制。Garthwaite 同时发现,当线程数量较少时(特别是当线程数量小于处理器数量时),每线程(per-thread)本地分配缓冲区的性能较好,而在线程数量较多的情况下,每处理器本地分配缓冲区的表现更佳,因此他们将系统设计成可在两种方案之间进行动态切换。

本地分配缓冲区通常使用顺序分配策略。每个线程(或处理器)也可以独立维护自身对应的分区适应空闲链表,同时使用增量清扫策略。线程在内存分配过程中会执行增量清扫,并将清扫所得的空闲内存单元添加到自身空闲链表中,但 Berger 等 [2000] 指出,如果将该

101

① 还有一些学者使用线程本地堆(thread-local heap)这一术语。在本书中,“本地分配缓冲区”这一术语主要强调的是线程之间相互隔离的内存分配,而“线程本地堆”这一概念在则偏重于相互隔离的分区回收。因此,“线程本地堆”几乎必然是“本地分配缓冲区”,反之则不一定成立。

算法用于显式内存管理会存在一些问题。例如，在某一使用生产者-消费者模型的程序中，消息对象通常由生产者创建并由消费者释放，因此两个线程之间将会产生单方向的内存转移。在垃圾回收环境下通常不会存在这一问题，因为回收器可以将空闲内存释放到全局内存池中。如果使用增量清扫，空闲内存单元将被执行清扫的线程所获取，从而自然地将回收所得的内存返还给分配最频繁的线程。

7.8 需要考虑的问题

对于垃圾回收系统中内存分配器的设计，有几个特别值得注意的问题：分配算法的设计绝不能独立于回收算法而进行；标记-清扫等非移动式回收器或多或少都会依赖基于空闲链表的分配策略，但在 10.3 节我们将看到与顺序分配相结合的实现方式；某些线程本地分配缓冲区也会使用顺序分配与标记-清扫回收相结合的分配策略；复制式与整理式回收通常使用简单且快速的顺序分配策略，与分区适应空闲链表分配相比，其速度较快，而更重要的是，顺序分配实现的简单性通常意味着更高的可靠性。

如果在使用标记-清扫策略的同时，偶尔或者在极端情况下进行堆整理以消除内存碎片，那么在整理完成后，回收器需要更新相关的分配数据结构以反映最新的内存分布情况。

使用额外的位图表来标记内存颗粒的状态（空闲/已分配）以及内存单元/对象的起始地址，不仅能提升程序的鲁棒性，而且可以简化对象头部的设计。该策略同时可以加速回收器的操作，并可以在存储器层次结构方面提升回收器的性能。基于位图的分配策略空间开销不大，但其分配过程通常会存在额外的时间开销。

基于内存块的分配可以在具体的编程语言实现（例如每个内存块中仅分配一种类型的对象）以及回收器元数据两个方面减少每个对象的开销，但这一优势可能会被内存块中的闲置空间以及无法使用的空间所抵消。对于使用多种不同的分配策略和回收策略来管理多个内存空间的场景，基于内存块的分配通常可以较好地发挥作用。

分区适应分配通常会比基于单个空闲链表的分配要快，对于垃圾回收系统而言，这一点尤为重要，因为基于垃圾回收的程序通常会比基于显式内存管理的程序使用更多的动态内存分配。

垃圾回收系统中的对象通常成簇回收，因此针对显式内存管理系统设计的相邻空闲内存单元合并策略便不再必要。标记-清扫回收可以在清扫阶段快速重建空闲链表。对于整理式回收器而言，一次整理完成后，堆中通常只会存在一个用于顺序分配的大内存块。复制式回收则会将一个半区整体释放，从而无需对单个内存单元进行回收。

堆内存的划分

到目前为止我们都假定垃圾回收以这样的方式进行：所有的对象是由相同的垃圾回收算法管理的，并且所有垃圾将在同一时间得到回收。然而这个假设并不是必须的，如果我们对不同的对象加以区别对待的话，在回收处理的性能上将得到相当大的好处。最广为人知的例子就是所谓“分代回收算法”(generational collection) [Lieberman and Hewitt, 1983; Ungar, 1984]，该算法将对象按不同的年龄 (age) 进行分隔，并优先回收更年轻的对象。有很多理由可以证明，对不同类别的对象进行差异化处理的方法是有益的。其中部分理由可能会与管理这些对象的回收器技术有关。正如前面章节中我们所看到的那样，对象的管理既可以采用直接算法 (如引用计数)，也可以采用间接算法，即追踪算法来完成。追踪算法可能会移动对象 (标记-整理算法或复制算法)，也可能不移动 (标记-清扫算法)。于是我们会考虑是否希望垃圾回收器移动不同类别的对象，如果是，如何移动它们为佳。我们可能希望能够依照对象的地址快速判断出应当对其使用何种回收或分配算法。最一般的情况是，我们可能希望区分何时回收不同类别的对象。

8.1 术语

区分那些我们想把某些特定的内存管理策略应用其上的对象集合是有用的，同时我们也可以使用一些机制来更有效地实现这些内存管理策略。我们将使用术语“空间”来表示那些使用相同处理方法的对象的逻辑集合。一个空间可能会使用一个或多个地址空间中的内存块。所有的块都是连续的，大小通常为 2 的整数次幂，并按 2 的整数次幂地址对齐。

8.2 为何要进行分区

将堆分割成几个分区，每个分区采用不同管理策略或不同机制的做法通常是有效的。这些想法最初出现在 Bishop 的那些很有影响力的论文 [1977] 中。使用内存分区处理的原因包括对象的移动性、大小、更低的空间消耗、更简单的对象性质识别、垃圾回收效率的改善、停顿时间的降低、更好地实现局部性等。在我们考虑特定的垃圾回收模型和利用堆内存划分方法来进行对象管理之前，我们先来检验一下这些原因。

103

8.2.1 根据移动性进行分区

在一个混合回收器 (hybrid collector) 中，识别哪些对象可移动，哪些对象不能移动或移动代价很大是十分必要的。当运行时系统和编译器之间缺乏沟通或者一个对象被传给了操作系统 (例如，一个 IO 缓冲区) 时，对象就很有可能无法移动。Chase [1987, 1988] 指出异步移动对象可能还会不利于编译器的优化。为了移动一个对象，我们必须能够找到所有指向这个对象的引用，以便将每一个引用改为指向对象的新位置。相反地，如果回收过程是不移动对象的，则追踪式回收器只需要找到至少一个引用就足够了[⊖]。因此，当一个引用被传给

⊖ 找到至少一个引用说明该对象不是垃圾。——译者注

一个根本不关心垃圾回收的程序库（例如，传给 Java 的原生接口）的时候，该对象就不能被移动了。此时，要么这个对象必须被钉住（pinned），要么我们必须保证对象在被程序库访问期间垃圾回收动作不能在其空间中执行[⊖]。

那些因为所指对象被移动而必须被更新的引用包含了根集合中的元素。然而确定根引用与待移动对象之间精确的映射关系对构建托管语言与其运行时环境之间的接口来说是一个更富挑战性的工作。我们将在第 11 章再详细讨论这块内容。最初实现的流程通常是保守地扫描根集合（线程栈和寄存器），而不是构建一个栈帧槽（stack frame slot）等与其所含对象引用之间的类型精确（type-accurate）的映射。当编译器（例如，C 和 C++ 编译器）不能提供精确类型信息时更是如此。保守式栈扫描 [Boehm and Weiser, 1988] 将每一个栈帧中的槽（slot）都当作一个潜在的索引项，并使用一些测试来丢弃那些不可能是指针的值（例如，那些超出堆内存范围的值或指向的位置没分配任何对象）。由于保守的栈扫描标明了栈中所含有的真实指针槽的超集，所以改变这些槽里的数据就是不可能的了（因为我们可能会在不经意之间更改某一整数，而这个整数可能刚好就是一个指针）。因此保守的垃圾回收算法不能移动任何被根直接引用的对象。然而如果可以适当地给出堆中对象的一些信息（可以不一定是完整的类型信息），一个主体复制（mostly-copying）垃圾回收器就可以安全地移动除了模糊根（ambiguous roots）集合 [Bartlett, 1988a] 直接可达的对象之外的任何对象。

8.2.2 根据对象大小进行分区

在某些情况下移动对象可能是不合适的（但并非不能移动）。例如，移动大对象的代价可能比因不移动它们所产生碎片的代价更大。一个常用的策略是，将大小超过特定阈值的对象分配到一个专门的大对象空间（large object space, LOS）里去。我们在之前的章节中已经看到过分区适应分配器是如何区别对待大对象和小对象的。大对象通常被放置在独立的内存页（所以大对象的尺寸至少应该是内存页大小的一半），并且通过一个像“标记-清扫”这样不移动对象的回收器来进行管理。值得注意的是，通过将对象放置到它们的专属内存页，我们既可以使用 Baker 的转轮（treadmill）回收器 [1992]，也可以通过重新映射虚拟内存页的方式 [Withington, 1991] 来实现虚拟“复制”。

8.2.3 为空间进行分区

104

对象隔离在减少堆空间的整体需求方面可能很有用。在一个由支持快速分配并提供良好空间局部性（当一个对象序列被分配并初始化的时候）的策略管理之下的空间中创建对象是十分可取的做法。Blackburn 等 [2004a] 证明连续分配和空闲链表这两种内存分配方法之间的代价差异是非常小的（只占总执行时间的 1%），两者之间真正的差异主要是由两种分配方法对局部性改进程度的二阶效应（second order effect of improved locality）决定的。特别是对年轻对象，如果把它们按照分配顺序排列，将会为其带来一些额外的好处。

复制和滑动回收器都可以消除碎片并允许顺序内存分配。但是复制式回收器需要的地址空间是非移动式回收器的两倍，而标记-整理回收方式则相对较慢。因此将对象进行隔离通常是有益的，这样便于让不同的内存管理器来管理不同的内存空间。对于那些生命周期较长并且内存碎片对其并非急需处理的对象来说，我们可以将其保存在一个主要是非移动操作、

⊖ 除了把对象引用直接传给程序库之外，将一个由垃圾回收器注册的、以便在必要时进行更新的间接引用（或句柄）传给程序库也是一个可选方案。这种方法现是 Java 原生接口的标准解决方案。

偶尔会进行某趟扫描来整理的空间里面。而那些分配频率、死亡率更高的对象则可以放到一个由分配速度快且回收代价小的复制式回收器管理的空间中去（相对于此类存活对象的数量比例来说，回收代价相对较低）。需要注意的是，为大对象预留复制空间的高昂代价是我们采用非复制式回收器来管理大对象空间的一个更深层次因素。

8.2.4 根据类别进行分区

将不同类别的对象进行物理隔离可以让“类型”这样的属性通过对象地址就可以简单地识别出来，而无需获取对象某一字段的值或更糟的是去追踪一个指针。除此之外，这样做还有以下几点好处。首先，这种做法充分利用了高速缓存的优势，因为它消除了加载更多字段的必要性（特别地，当某一特定类别对象的位置为静态时，地址之间的比较可对应于一个编译期的常量）。其次，通过属性来进行对象隔离，把所有共享相同属性的对象放置在相同的连续内存块中，我们就可以对空间进行基于地址的快速识别，同时此举也使得空间和对象属性关联了起来，而不用将相同的属性值在每个对象的头部复制一份。最后，对象的类别对于某些回收器来说是非常关键的。那些不含指针的对象是不需要被追踪式回收器扫描的。鉴于处理一个大型指针数组的开销很可能是由移动指针而不是像移动对象这样的开销决定的，所以将大型的与指针无关的对象保存在它们自己独立的空间里是有益的。当大型压缩位图（compressed bitmaps）成为伪指针（false pointers）的一个经常性来源时，如果把这些位图放到一个永远也不会被扫描的区域，保守式垃圾回收器会因此受益 [Boehm, 1993]。如果将那些不可能成为垃圾环路（garbage cycle）备选根的天生非循环（inherently acyclic）对象隔离保存的话，可回收循环引用垃圾的追踪回收器将因此获益。

虚拟机通常在堆内存中生成并保存代码序列（code sequence）。移动和清理代码会有一些特殊的问题，如确定和保持一致性、代码的引用或确定代码何时不再使用并因此可以被卸载（注意类的重新载入通常并不透明，因为类可能是有状态的）。代码对象也通常是体积庞大且长寿的。由于这些原因，不迁移代码对象通常是可取的做法 [Reppy, 1993]，同时也可将卸载代码作为针对特定应用程序的特殊情况来考虑。

8.2.5 为效益进行分区

进行对象隔离最著名的理由是要充分利用对象统计信息。某些对象从构建出来开始到程序结束一直被使用而另一些对象的生命周期却很短，这种现象很常见。早在 1976 年，Deutsch 和 Bobrow 就发现“统计显示，新分配的数据很可能是要么被‘敲定’，要么在相对较短的时间内被抛弃”。在 Java 程序中，一种十分普遍的现象是大多数分配点（allocation point）所创建的对象的生命周期符合双峰（bimodal）寿命分布 [Jones and Ryder, 2008]。大量研究已经证实，很多（而非全部）应用程序中对象的生命周期特征满足弱分代假说（weak generational hypothesis），即“大多数对象都在年轻时死亡” [Ungar, 1984]。无论是分代还是准分代（quasi-generational），众多垃圾回收策略的着眼点是将回收的重点集中到那些最有可能成为垃圾的对象上面，以达到花费最少的努力来尽量多地清理存储空间的目的。

如果对象生命周期的分布是足够偏斜的，则反复清理一个（或多个）堆内存的子集而非整个内存就是值得的 [Baker, 1993]。

例如，分代的垃圾回收器通常在每次回收整个堆之前就已经对堆中的某一空间（新生代或幼儿区）进行了多次回收。需要注意的是，这里面有一个权衡的考虑：由于不用在每次回收过程中都追踪整个堆，所以回收器将允许一些垃圾不被清理（在堆中漂浮）。这就意味着，

分配新对象时系统的可用空间是小于其应有大小的，因此垃圾回收器也会比在理想情况下调用得更频繁。更进一步，正如我们稍后会看到的那样，将堆内存分隔为已回收空间和未回收空间将会使赋值器和回收器增加更多簿记管理的负担，但倘若被选为进行回收操作的空间中的对象存活率足够低的话，分隔回收策略可能会非常有效。

8.2.6 为缩短停顿时间进行分区

对于一个追踪式回收器来说，回收操作的代价主要取决于需要进行追踪的存活对象的数量。如果使用复制式回收器，则清理操作的代价仅依赖于存活对象的数量，甚至在标记-清扫回收器中，追踪操作的代价也远超清扫操作。通过限制回收器需要追踪的定罪空间（condemned space）的大小，我们就可以限定需要清理或标记的对象的数量，从而控制垃圾回收所需的时间。在一个需要万物静止的回收器中，这样做就意味着可以缩短停顿时间。不幸的是，回收堆内存的一个子集改善的仅仅是预期的时间，因为对某个单一的空间回收之后所返回的空闲内存可能仍无法让计算过程继续进行，所以还是需要回收整个堆。因此，一般而言，分区回收不能降低最坏情况下的停顿时间。

在某些极端情况下，分区策略可以使一个空间能够在常数时间内被清理完。如果一个定罪区域中的所有对象从该区域之外都是不可达的，则对于回收器来说，清理该区域时不需要进行任何追踪工作：该区域所占内存可以被一并返回给分配器。要判定一个区域是不可达的，需要将合适的对象访问规则与堆结构（例如有界区域的栈）有机地结合起来。正确使用责任通常全都落在程序员身上（正如 Java 的实时规范中定义的那样）。然而，如果能给一个类似 ML 这样合适的语言，这些区域还是可以被自动推断出来的 [Tofte and Talpin, 1994]。通过一个允许对某些类型和区域引用进行推断的复杂的类型系统，C 语言的扩展 Cyclone 可以有效地减轻程序员的负担 [Grossman 等, 2002]。

8.2.7 为局部性进行分区

随着内存层级变得日益复杂（更多的层级、多 CPU 核心、多插槽，以及非一致性内存访问），局部性对于性能的重要性不断增加。简单的回收器与虚拟内存和高速缓存的交互往往比较差。作为追踪动作的一部分，追踪式回收器需要接触每一个存活对象。标记-清扫回收器同样可能需要接触已经死亡的对象。复制式回收器则可能会接触每一个堆内存页，即使在任意时间点只有一半的内存是用来保存存活对象的。

[106]

研究人员长期争论的一个问题是，垃圾回收器不应该只是简单地用来清理垃圾，还应该用来改善整个系统的局部性 [Fenichel and Yochelson, 1969 ; White, 1980]。我们在第 4 章已经看到如何改变复制式回收器的遍历顺序，以便通过父子对象共置的方式提高赋值器的局部性。分代回收器可以使回收器和赋值器的局部性都得到进一步的改善。该回收器得益于将大部分精力集中在堆的某一个分区上，这种做法有利于用最小的代价获取最多的空闲空间。减少工作集的大小对于赋值器也是有益的，因为年轻对象的变化率通常比老对象要高 [Blackburn 等, 2004]。

8.2.8 根据线程进行分区

垃圾回收动作需要在赋值器线程和回收器线程之间进行同步。对于每次只需中止不超过一个赋值器线程的即时回收（on-the-fly collection），其可能需要一个与其他赋值器线程进行握手同步的复杂系统才能实现，而即使是万物静止的回收算法也需要通过同步来中止所

有赋值器线程的运行。如果我们每次只中止一个线程，并仅回收由该线程分配的且尚不能从其他线程可达的那些对象，则同步代价是可以减少的。为了做到这一点，回收器必须能够区分出哪些对象是仅单个线程可访问的而哪些对象是被多个线程共享的，例如可以使用线程本地子堆（heaplet）进行分配 [Doligez and Leroy, 1993 ; Doligez and Gonthier, 1994 ; Steensgaard, 2000 ; Jones and King, 2005]。

在一个更大的粒度上，识别访问特定任务的对象可能是有用的，其中的“任务”是由一个互相协作的线程集合组成的。例如，一个服务器上可能有多个应用程序在运行，而每一个应用程序通常要求它们自己的虚拟机整个被载入并初始化。与此相反，多任务虚拟机（multi-tasking virtual machine, MVM）则允许多个应用程序（任务）在单个多任务虚拟机中运行 [Palacz 等, 1994 ; Soman 等, 2006、2008 ; Wegiel and Krintz, 2008]。我们需要小心谨慎，以确保不同的任务不能直接（访问其他任务的数据）或间接地（拒绝其他任务对如内存、CPU 时间等系统资源的公平访问）影响对方。若能既干扰其他任务（例如，不需要运行垃圾回收），又能在完成之后将与本任务相关的所有资源释放掉就更好了。通过将属于不同线程的非共享数据进行分隔，所有这些问题的处理都能得到简化。

8.2.9 根据可用性进行分区

我们不希望接触其他线程可达对象的一个原因是为了降低同步开销。然而，我们可能也希望将对象按用途分区，因为回收器可能会根据对象的物理分布采取不同的处理策略。Xian 等 [2007] 观察到，在一个应用程序服务器中，作为客户端请求的一部分而初始化的远程对象往往比本地对象存活的时间更长；扩展 Sun 的 HotSpot 分代回收器来识别并专门处理这些对象，可以有效提高服务器的工作负载。更一般地，在一个由分布式垃圾回收管理的系统中，最好能够用不同的策略和机制来管理远程和本地的对象与引用，因为访问远程对象的代价比访问本地对象大几个数量级。

分布式并不是导致对象访问代价不一致的唯一因素。之前我们花了很大精力研究追踪式回收器如何最小化高速缓存不命中的代价。高速缓存不命中的代价可能是几百个时钟周期，而访问一个所在内存页被交换出去的对象代价将是几百万个时钟周期。对于一个早期的分代回收器来说，避免频繁缺页是需要优先解决的问题，时至今日，导致频繁换页的配置可能会被认为是在进行不可救药的破坏^①。物理页的组织（在内存中或换出了）可以被认为是另一种形式的堆分区，是一个切实可资利用的特性。书签（Bookmarking）回收器 [Hertz 等, 2005] 通过与虚拟内存系统配合协作，来改进对于被换出页的选择（从回收器视角），并使回收器可以在无需访问非驻留内存页里对象的情况下完成追踪操作。

[107]

同样地，非一致内存访问的计算机有一些内存银行（memory bank），这些内存银行和某些处理器挨得比较近。Sun 的 Hotspot 回收器可以识别这一属性，并优先在“附近的”内存中分配对象，对于那些访问不同内存区域耗时差异显著的大型服务器，这一策略可以最大限度地降低内存访问时延。

8.2.10 根据易变性进行分区

最后，我们可能希望根据对象的易变性来对其进行分区。相较于存活时间较长的对

① 另一方面，很多新一代上网本的内存都是很有限的，因此频繁换页是一个需要仔细关注的问题。

象,那些新创建的对象往往被修改的更频繁(如初始化其属性字段)[Wolczko and Willams, 1992; Bacon and Rajan, 2001; Blackburn and McKinley, 2003; Levanoni and Petrank, 2006]。基于引用计数的内存管理往往会产生很高的每次更新(per-update)开销,因此不大适合管理那些更新频繁的对象。另外,在非常大的内存堆中,可能在任意时间周期内都只有相对一小部分对象被更新,但一个追踪式回收器却必须访问所有的垃圾候选对象。引用计数在该场景下可能更为适用。

Doligez 和 Gonthier 通过易变性(和线程原则)将 ML 对象分隔开来,他们为每个线程配备了私有的不可变对象空间、非共享对象空间,同时所有线程共享同一个空间[Doligez and Leroy, 1993; Doligez and Gonthier, 1994]。他们的模式需要一个关于引用的很强的属性:线程本地堆外(其他线程的本地堆或共享空间)的指针没有指向该堆里的任何对象。通过把更改的内容用写时复制(copy on write)的方法同步到共享内存区的策略,我们成功避免了线程本地堆内存被外部引用。这在语义上是透明的,因为引用的目标是不可变的。综上所述,这些属性让每个线程的私有堆都可以被异步回收。该方法的一个额外的优势是,与其他大多数各自独立回收每个空间的策略不同,该方法不需要追踪跨空间的指针(虽然赋值器仍必须检测这些指针)。

8.3 如何进行分区

最直观、普通的分区方法大概就是将堆内存分成互不重叠的地址段。在最简单的情况下,每个空间占据连续的堆内存块,因此这种映射是一对一的。把这些内存块按 2 的整数次幂地址边界对齐将是更为有效的做法。在这种情况下,对象所属的空间信息就相当于被编码到地址的最高位,并可以通过移位或掩码操作来找到具体位置。一旦获知了空间的特性,回收器就可以决定如何处理其中的对象了(例如,标记、复制、忽略等)。如果能在编译期获知空间的布局(layout),则这种尝试(即与一个常量进行比较)可能会特别有效。此外,若把这些二进制位看成内存块表的索引,我们就可以对空间进行查找操作了。

然而,对 32 位操作系统的内存使用来说,将内存分割成连续的内存区可能不是最有效的方案,因为这些区域占据的虚拟地址空间必须是事先保留的。虽然那些即时的划进/划出(mapped in and out)连续空间的操作并不会提交物理内存页,但连续的地址空间或多或少会感觉不大灵活并可能导致虚拟内存耗尽,即使系统中仍有足够的物理内存页可用。还有一个额外的困难是,在很多情况下操作系统都趋向于把类库的代码段映射到不可预知的地方——有时是故意为之,以便于改善系统的安全性。这使得预留大块连续虚拟地址空间的操作变得非常困难。在大部分情况下,改用 64 位地址空间就可以有效解决这些问题。

另一种方法是将空间实现为非连续内存块的地址空间集合。不连续空间通常都包含几组连续地址空间,而每段地址空间又是由固定大小的帧组成的。如前所述,如果帧地址是依照 2 的整数次幂边界对齐,且为操作系统页大小的整数倍,则对于帧的操作将会更高效。同样地,这样做的缺点是在获取对象的空间信息时需要查表。

通过把对象在物理上分隔开以实现特定空间的做法并不总是必要的。相反,对象所属的空间可以通过在其头部添加若干位的方式来加以标识[Domani 等, 2002]。虽然这使我们无法通过一个快速的内存比较来对空间进行判定,但这种方法或多或少还是有一些优点的。首先,这种方法允许我们根据像年龄或线程可达性这样的运行时属性来进行对象划分,即使是在回收器不移动对象的情况下也适用。其次,该方法可能会有助于处理那些需要被暂时钉住

的对象（例如那些代码可访问但回收器没察觉到的对象）。最后，运行时的动态分区可能会比静态分区更精确一些。例如，静态逃逸分析（escape analysis）^①提供了一个对象是否可能被共享的保守估计。虽然 Hones 和 King[2005] 说明了如何在线程本地分配场景中获得更为精确的静态逃逸估计（estimate of escapement），静态分析还是不能适应超大型程序的需求，而且动态类加载技术的出现通常迫使该分析更加保守化。如果对象的逃逸是被动态追踪的，则当前线程本地对象和那些正在（或已经）被超过一个线程访问的对象之间的差别就很明显了。动态分隔的缺点是会引入更多的写屏障（write barrier）。一旦一个指针更新操作使其所指对象成为潜在的共享状态，则其所指对象和传递闭包都必须被标记为共享。

在这一小节的最后，我们注意到仅回收内存堆分区的一个子集必然会导致回收器回收不完全：回收器无法回收位于尚未进行回收的分区里的任何垃圾。即使垃圾回收器在一段时间之内仔细清理每一个分区，假定存在环状引用，那跨越多个分区的垃圾对象环仍不能被回收。为了保证回收的完整性，必须在待回收分区以及未被清理的对象被移动到的目的分区上施加一些规则。一个简单且广泛使用的解决方案是，当其他策略失败时，就回收整个堆。然而，当后面我们在考虑成熟对象空间（mature object space，也称火车回收器）[Hudson and Moss, 1992] 时，将会学到一些更为复杂的策略。

8.4 何时进行分区

分区的决策既可以静态（在编译期）完成，也可动态进行——当一个对象被创建时，或者在垃圾回收期间，或者当赋值器访问对象的时候。

最著名的分区方案是按对象的分代（generational）进行划分，在此方案中，对象根据其年龄被分隔开来，但这只是根据对象年龄的相关属性进行分区的形式之一。与年龄相关的回收器根据对象的年龄将其分隔成若干个空间。在这种情况下，分区是由回收器动态执行的。当一个对象的年龄增长超过某个阈值的时候，该对象将被提升（从物理上或逻辑上移动）到下一个分代的空间中。

[109]

由于在移动对象方面存在一些限制，将对象进行分隔的操作可能是由回收器来完成的。例如，当某些对象被钉住时，大多数复制回收器可能无法移动它们——这些对象被某些不会意识到它的位置可能会变动的代码访问。

分区的决策也可能是由分配器完成的。最常见的情况是，分配器会根据一个分配请求所需的内存大小来决定对象是否应该被分配到大对象空间中。在系统支持的对程序员可见的显式内存区或可以由编译器推测得到的区域（如作用域）内，分配器或编译器可以将对象放置到一个指定的区域中去。除非被明确告知对象是共享的，否则位于线程本地系统内的分配器会把对象放到该执行线程的本地子堆中。一些分代系统会尝试把一个新对象和指向它的那些对象置于相同的区域里，因为无论如何该对象最终都会被提升到那里 [Guyer and McKinley, 2004]。

通过对象的类型、代码或者一些其他的分析，对象所属的空间可以被静态地划定。如果可以提前预知某种类型的对象都存在某种公共属性，如永生性（immortality），则编译器就可以决定应该将这些对象分配到哪个空间，并生成适当的代码序列。分代垃圾回收器通常在为新对象预留的一块新生代区域中创建对象，随后，回收器将会把其中一些对象提升到更

① 逃逸指对象脱离了原线程本地内存的范围，成为了共享对象。——译者注

老的分代中去。然而，如果编译器“知道”某些对象（例如，那些在代码中某一特定地点创建的对象）通常会得到提升，则编译器可以把它们直接预分配（pretenure）到年老代里面去 [Cheng 等，1998；Blackburn 等，2001，2007；Marion 等，2007]。

最后，当堆内存被并发回收器（见第 15 章）管理的时候，对象还可以被赋值器重新划分。赋值器访问对象的操作可能会被读屏障或写屏障居中调节（mediated），这些都会导致一个或多个对象被移动或标记。对象的颜色（黑色、灰色、白色）以及持有对象的新/旧空间都可以被认为就是一个分区。分配器可以根据其他一些属性来动态地区分对象。如上所见，当对象逃逸出它诞生的线程时，Domani 等人 [2002 年] 使用的写屏障就可以把它们从逻辑上分隔开来。通过与操作系统进行协作，运行时系统可以在对象所在页被换入、换出时将对象进行重新划分 [Hertz 等，2005]。

在接下来的两章里，我们将会研究各种类型的分区式的垃圾回收器。第 9 章将会仔细研究分代垃圾回收器，而第 10 章将会检验大量其他模型，包括基于年龄或者其他非临时属性的多种分区策略。

分代垃圾回收

垃圾回收器的主要目的是找到已经死亡的对象并回收它们所占用的空间。在对象数量较少的情况下，追踪式垃圾回收器（特别是复制式垃圾回收器）能够最高效地进行回收。但长寿对象的存在却会影响回收效率，因为回收器不是反复地对其进行标记、追踪，就是反复地把它们从一个半区复制到另一个半区。第3章提到，在标记-整理回收器中，长寿对象会积累在堆底，并且回收器通常会避免对这些底部的“沉积物”进行处理。虽然这一策略可以减少移动长寿对象的次数，但是回收器仍然需要对它们进行扫描，也需要对它们所包含的引用进行更新。

分代垃圾回收是对上述算法的进一步改进与提升。在任何时候，分代垃圾回收算法都尽可能少地去处理长寿对象。弱分代假说（weak generational hypothesis）告诉我们，大部分对象都在年轻时死亡，因此可以利用这一特性尽量提高回收效益（即回收所得到的空间），同时减小回收开销。分代垃圾回收算法依照对象的寿命将其划分为不同的分代（generation），同时将不同分代的对象置于堆中不同的区域。回收器会优先回收年轻代对象，并将其中寿命够长的对象提升（promote）到更老的一代。

大多数分代垃圾回收算法通过复制的方法来处理年轻代对象。如果被处理的分代中存活对象的数量足够少，则其标记/构造率（mark/cons ratio）会降低，即回收器在一次回收过程中所需要处理的数据量与分配器在相邻两次回收之间所分配的数据量之间的比例较低。回收器处理年轻代对象所需要的时间主要取决于年轻代的整体大小，因此，回收某一分代的期望回收停顿时间可通过调整分代大小的方式来实现。在现有硬件条件下，一个配置合理的垃圾回收器（在运行符合弱分代假说的程序时）完成一次年轻代垃圾回收所需的时间通常是10ms级别。假定两次垃圾回收的时间间隔足够长，则该类型垃圾回收器可以满足绝大多数应用程序的要求。

某些情况下，分代垃圾回收器需要对整个堆进行回收。例如可用内存已经耗尽，或者仅仅依靠回收年轻代所能获取的内存已经不足。因此分代垃圾回收器最多只能改善期望停顿时间（而非最大停顿时间），这显然不能满足实时系统的要求。第19章讨论硬实时系统下的垃圾回收算法。

分代垃圾回收可以通过减少对年老代对象的处理次数来提升内存吞吐量，但这需要付出一定的额外开销。我们知道，仅针对年轻代对象的回收并不能回收任何一个年老代垃圾对象，同时回收器也无法及时回收已经成为垃圾的长寿对象。因此，为了能够独立回收某一分代，分代回收器必须在赋值器之上维护一个额外的记忆集来记录分代间指针。与分代回收带来的收益相比，这一开销是值得付出的。如何设计分代垃圾回收器才能同时达到提升吞吐量与减少停顿时间的目的，是一门精妙的艺术。

[11]

9.1 示例

图9.1是分代垃圾回收的一个简单示例，它使用两个分代。新对象诞生在年轻代。在每

一次年轻代对象回收过程中，如果某一对象的寿命够长，则回收器会将其提升到年老代。在进行第一次回收之前，年轻代包含 4 个对象：N、P、V、Q；年老代包含三个对象：R、S、U。对象 R 和 N 同时被堆之外的根或者其他对象引用。假设对象 N、P、V 已经存活了一段时间，但是对象 Q 却是在回收过程开始之前刚刚创建的。此时，将哪个年轻代对象提升到年老代是一个十分重要的问题。

分代垃圾回收器的一个重要任务便是把寿命够长的年轻代对象提升到年老代。要达到这一目的首先需要一种测量对象寿命的方法，并通过某种方法将其记录。在图 9.1 中的年轻代对象里，对象 N 直接被根对象引用，对象 P、Q 通过年老代对象 R、S 间接地被根对象引用。绝大多数分代垃圾回收器不会扫描整个堆，而只会扫描正在进行回收的一代中的对象。为达到这一目的，分代垃圾回收器必须额外维护一个分代间指针集合，例如，该集合会记录对象 S，通过该集合便可快速找到对象 P 和 Q。

有两种情况会导致分代间指针的出现，一是赋值器在一个年老代对象中写入一个指向年轻代对象的指针，二是回收器将一个年轻代对象提升到年老代，如图 9.1 中将对象 P 而 Q 提升到年老代。不论哪种情况，都可以通过一个写屏障（write barrier）来检测分代间指针的产生。也就是说，为捕获分代间指针赋值器，必须通过写屏障来进行指针写操作，回收器也必须依赖一个简单的复制写屏障来对提升过程中出现的分代间指针进行探测。在图 9.1 所示的例子中，所有包含对回收过程有用的分代间指针的对象（即：S、U）都会被记录在记忆集（remset）中。

不幸的是，将分代间指针作为次级回收根集合的方法加剧了浮动垃圾问题：尽管次级回收的频率较高，但其无法对年老代对象进行回收（如对象 U）。更糟糕的是，U 持有一个分代间指针，这导致回收器必须将其当作年轻代的根集合来对待。这一“庇护”（nepotism）效应将导致回收器把对象 V（即对象 U 的后代）提升到年老代（而不是将其回收），从而进一步减少了年老代的可用空间。

9.2 时间测量

将对象依照寿命进行分代的前提是如何去测量对象的寿命。对象寿命的测量有两种方式，一种是基于对象所经历的时间，另一种是基于堆所分配的字节数。墙上时钟对于理解系统的外部行为是有用的，程序运行的总时间是多少？垃圾回收过程的停顿时间是多少？停顿时间如何分布？这些问题的答案决定了系统是否满足预定的设计目标：是需要足够多的时间去执行任务，还是需要足够快的响应速度。后者既是人机交互系统的要求，也是硬实时系统（如嵌入式系统）或者软实时系统（允许少量的交互延迟）的要求。然而，确定对象寿命的更好方法是统计该对象存活期间堆所分配的字节数。尽管 64 位系统的指针和整数会比 32 位系统占用更多空间，但空间分配在很大程度上仍是一种不依赖机器的测量方法。字节分配数同时也直接反映出内存分配器的压力，这一压力很大程度上与垃圾回收过程的频率相关。

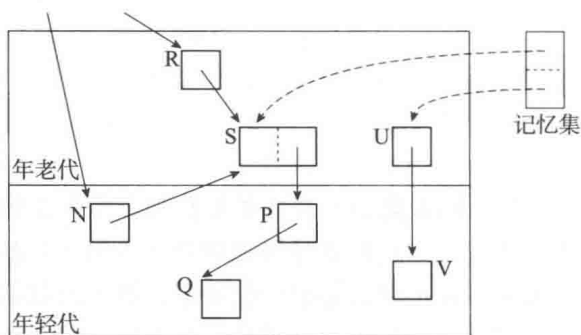


图 9.1 分代间指针。如果想要在年轻代回收过程中避免对年老代的扫描，就需要通过一个特殊的数据结构和机制来记录年老代中有哪些对象引用了年轻代对象（S 和 U）

不幸的是,在多线程系统下(此时系统中包含多个程序或系统线程),以堆分配的字节数来测量寿命存在一定的困难:计数器会受到一些与垃圾回收过程无关的内存分配的影响,所以如果简单地统计整体内存分配情况,测量结果可能偏大 [Jones and Ryder, 2008]。

实际应用中的分代垃圾回收器一般会将对象所经历的垃圾回收次数作为该对象的寿命,这样不仅记录更加方便,而且占用的额外空间较少,但这只是字节分配测量方法的一个近似替代。

9.3 分代假说

弱分代假说 (weak generational hypothesis) 的含义是:大多数对象都在年轻时死亡。这一假说已经在各种不同种类的编程范式或者编程语言中得到证实。Federaro 和 Fateman[1981]发现,在一个基于 MacLisp 的计算机代数系统中,98% 的对象会在其所经历的首个垃圾回收过程中得到回收;Zorn[1989]声称,Common Lisp 语言中 50% ~ 90% 的对象寿命不超过 10KB。这些规律在函数式编程语言中都是相似的:在 Haskell 语言中,75% ~ 90% 的对象寿命不超过 10KB,而只有 5% 能活过 1MB[Sansom and Peyton Jones, 1993];Appel[1992]发现,在标准 ML/NJ 语言中,对于要处理的分代,每次回收过程可以回收 98% 的对象,Stefanović 和 Moss[1994]进一步发现,只有 2% ~ 8% 的对象可以活过 100KB 的阈值。

[113]

这一假说在基于面向对象语言的程序中依然成立。Ungar[1986]发现,Smalltalk 中只有不到 7% 的对象可以活过 140KB;Dieckmann 和 Hölzle[1999]声称,在 SPECjvm98 基准程序套件中,1% ~ 40% 的对象可以活过 100KB,只有不到 21% 的对象可以活过 1MB,这一比例会因为应用程序的不同而产生较大差异;Blackburn 等 [2006]发现,在 SPECjvm98 和 DaCapo 基准程序套件中,年轻代平均只有不到 9% 的对象可以活过 4MB,尽管不同基准程序得出的测试结果差别较大,但是这一数字却是一个普遍的上限,因为这一数字包含在最后一次回收过程之前新创建的对象;Jones 和 Ryder[2008]发现,在 Java 应用程序中,对象寿命通常符合双峰 (bimodal) 分布,即 65% ~ 96% 的 Dacapo 对象活不过 64KB,只有 3% ~ 16% 的对象可以活过 4MB。即使在没有自动内存管理功能的命令式语言 (imperative language) 中,大多数对象的寿命也通常较短:Barrett 和 Zorn[1993]声称,50% 以上的对象活不过 10KB,10% 以下的对象能活过 32KB。

与弱分代假说相比,支持强分代假说 (strong generational hypothesis) 的证据则稍显不足。强分代假说的含义是:越老的对象越不容易死亡 [Hayes, 1991]。像弱分代假说这样的简单模型能够很好地描述许多应用程序中对象的整体寿命规律,然而一旦短命对象的比例减少,较大时间范围内对象寿命的分布模型就会更加复杂。程序的处理通常都是分成一个个阶段 (phase) 的,因而对象的寿命不是随机的,它们通常是成簇出现且成批死亡的 [Dieckmann and Hölzle, 1999; Jones and Ryder, 2008]。有相当一部分对象会一直存活到程序结束。对象的寿命也可能与它们的大小存在一定联系,尽管这一观点还存在争论 [Caudill and Wirfs-Brock, 1986; Ungar and Jackson, 1988; Barrett and Zorn, 1993],但对大型对象进行特殊处理也具有一定意义。

9.4 分代与堆布局

回收器可以使用多种不同的策略来组织对象的分代。它们可以在物理上或者逻辑上使用两个或更多分代,也可以将所有分代限制在相同大小的空间内,还可独立维护每个分代的空

间大小；分代内部的数据结构可以是扁平的 (flat)，也可以是一系列基于对象寿命的子空间，称之为“阶”(step) 或者“桶”(bucket)；分代内部可以包含存放大对象的特定子空间，每个分代也可以使用不同的算法来维护其中的对象。

分代垃圾回收器的主要设计目标是减少回收过程的停顿时间，同时提升空间吞吐量。如果使用复制的方法对年轻代对象进行回收，那么期望的停顿时间很大程度上取决于次级回收 (minor collection) 之后的存活对象总量，而这一数值又取决于年轻代的整体空间大小。如果年轻代的整体空间太小，那么虽然一次回收过程很快，但两次回收的间隔过短，年轻代对象没有足够的时间到达死亡，因而回收到的内存不多，这一情况将引发很多不良后果。

首先，没有足够多的对象可以在短时间内死亡，进而导致年轻代对象的回收过于频繁，且存活下来需要复制的对象数量变多。频繁的垃圾回收会增大回收器停顿线程、扫描其栈上数据的开销。

114

其次，将较大比例的年轻代对象提升到年老代会导致年老代被快速填充，进而增大年老代，甚至整个堆的垃圾回收频率。另外，过早地将年轻代对象提升到年老代还会导致“庇护”现象的出现：年老代中的垃圾会使得它们的年轻后代在次级回收中存活下来，所以回收器会将这些年轻后代提升到年老代，从而进一步提高了提升率。

再次，许多证据表明，对新生对象的修改会比对年老对象的修改更加频繁。如果过早地将年轻代对象提升到年老代，则大量的更新操作 (mutation) 会给赋值器的写屏障带来较大压力，这种现象是不希望出现的。因此我们必须结合系统的真实负载来平衡赋值器和垃圾回收过程的开销。在一个设计良好的系统中，垃圾回收所占用的运行时间一般都小于赋值器。例如，对于某个快速路径 (fast path) 中只包含少量指令的写屏障，假设其占用整体运行时间的 5%，进一步假设垃圾回收占用整体运行时间的 10%，其他种类的写屏障则很容易将拦截过程占用的时间翻倍，即额外增加 5% 的负载。为了补偿写屏障带来的额外负载，垃圾回收器必须将自己的时间消耗减半，但这是很难实现的。

最后，对象的提升将会使程序的工作集合变得稀疏。因此，分代垃圾回收器的设计是一门对这三方面进行平衡的艺术：不仅要尽量加快次级回收的速度，而且要尽量减少次级回收以及成本更高的主回收 (major collection) 的频率，最后还应当尽量减少赋值器的内存管理开销。下面我们将介绍如何达到这一目标。

9.5 多分代

如果回收器使用更多的分代，不仅可以快速回收年轻代，而且可以降低年老代的填充速度，进而降低整个堆的回收频率。中间代能够筛选出那些经历过年轻代回收后仍然存活但很快就会死亡的对象。如果回收器将年轻代回收的存活对象集体提升，那么其中将包含很多刚刚诞生但在不久的将来就会死亡的对象，而如果使用多分代垃圾回收，不仅可以使年轻代保持较小的整体大小以减少回收停顿时间，而且还可以避免最老代中出现一些很快就会死亡的对象。

多分代垃圾回收器也存在一些缺陷。大多数系统在回收最老代对象的同时也会回收年轻代，这带来一个好处，即需要记录的分代间指针就会只有一个方向，即从年老代到年轻代，这一方向的引用一般比反向的引用要少。尽管对中间分代进行一次回收的时间小于回收整个堆所需要的时间，但是仍然会大于一次年轻代回收所需要的时间。多分代垃圾回收器的实现通常比较复杂，而且会给回收器的扫描过程带来额外的负担，其关键实现代码也与两分代垃

圾回收器存在一定的差别（两分代垃圾回收器通常可以用一个固定的地址来进行分代，甚至可以是一个编译期决定的地址）。分代的增多会导致分代间指针相应增多，进而给赋值器的写屏障带来更大压力（当然这取决于具体的实现）。另外，中间分代的出现将创建更多的分代间指针，从而增大年轻代的根集合。

很多为 Smalltalk 和 Lisp 设计的早期分代垃圾回收器都使用多分代回收机制，但现代面向对象语言所依赖的分代垃圾回收器大部分都仅使用两个分代。一些分代垃圾回收器（例如在分配率和死亡率都非常高的函数式语言中所使用的回收器）尽管提供多于两分代的回收机制，但是在默认情况下仍然仅使用两分代 [Marlow 等, 2008]。不同分代的处理机制（特别是最年轻代的处理机制）可以用来控制对象的提升率。

115

9.6 年龄记录

9.6.1 集体提升

对象的年龄记录方式与回收器的提升策略紧密相关。多分代是记录对象年龄的一种粗略方法。图 9.2 展示了对年轻代进行组织以控制对象提升的四种策略，我们将逐一进行介绍。最简单的组织方式是将年老代之外的其他分代当作独立的半区（见图 9.2a），当回收器对某一分代进行回收时，直接将其中的存活对象集体提升到下一代。该方案不仅实现简单，而且各年轻代的内存空间得到最大程度的利用，因为回收器不仅无需单独记录每个对象的年龄，也无须为每一代预留专门的复制保留区（最老代使用复制式回收策略的场景除外）。Jikes RVM Java 虚拟机中，MMTk 内存管理器所用的分代式回收器便采用这一集体提升方案 [Blackburn 等, 2004b]。但 Zorn[1993] 指出，（在 Lisp 系统中）与仅提升经历多轮次级回收后依然存活的对象策略相比，集体提升策略的提升率要高出 50% ~ 100%。

图 9.3 [Wilson and Moher, 1989b] 展示了年轻代对象所经历的回收次数（一次或两次）与其存活率之间的关系。在“大多数对象都在年轻时死亡”这一前提下，图中的曲线展示了 t 时刻分配的对象在经历数次回收之后依然存活的比例。从中我们可以看出，对象的创建时间越接近下一轮回收，则其在下一轮回收中存活的几率就越高。我们将注意力集中在第 n 次和第 $n+1$ 次回收之间的图形区域。曲线 (b) 展示了经历一轮回收后依然存活的对象的比例。

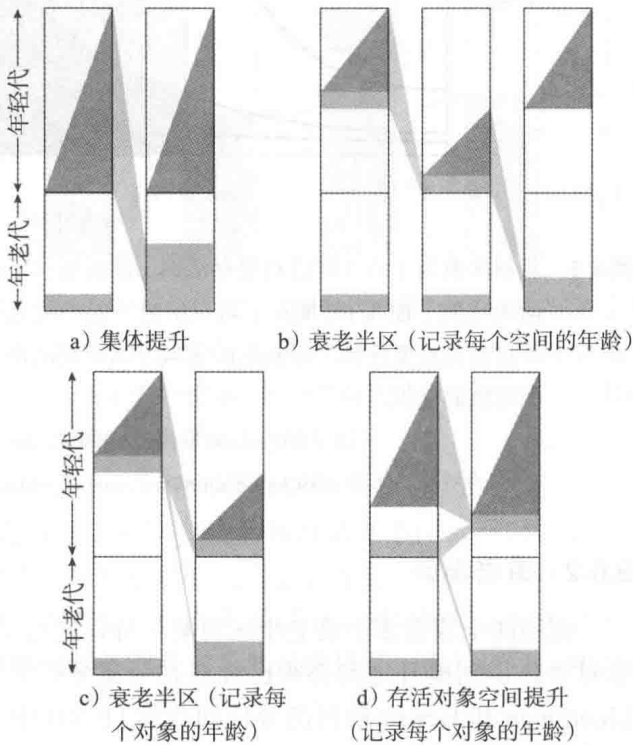


图 9.2 分代垃圾回收器中的半区组织方式。深灰色表示新分配的对象，浅灰色表示已完成复制或者已得到提升的对象。每种情况下，x 轴代表时间，y 轴代表现有对象的分布

见 Jones[1996] 一书，经许可后转载

率,从中我们可以看出,大多数对象活不过其所经历的第一轮回回收,即图中的浅灰色区域。位于曲线(c)下方黑色区域中的对象可以经历两轮回收。如果我们将一次回收之后所有的存活对象集体提升,则曲线(b)之下的深灰色区域以及黑色区域中的对象都会得到提升,但如果仅提升经历两次回收的对象,则只有曲线(c)下方黑色区域中的对象才会得到提升。如果每个对象都包含一个上限大于1的复制计数器,则回收器可以避免提升过于年轻的对象(它们可能很快死亡),从而显著降低提升率。如果将提升的标准设置为大于两次回收,反而可能起到负面效果[Ungar, 1984; Shaw, 1988; Ungar and Jackson, 1988]。Wilson[1989]指出,如果要提升率降低一半,则对象在得到提升之前所需经历的复制次数可能会增加四倍或者更多。

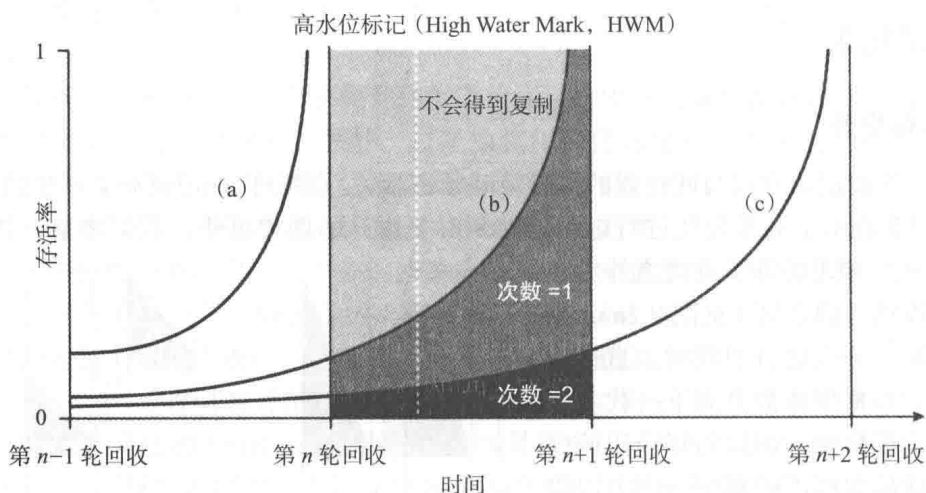


图 9.3 复制次数为 1 或 2 时的对象存活率。曲线展示了诞生于 x 时刻的对象在未来的垃圾回收过程中存活的比例。曲线 (b) 展示了可以活过一轮回收过程的对象比例，曲线 (c) 展示了可以活过两轮回收过程的对象比例。带颜色的区域可以反映出在不同复制次数策略下最终得到提升或者未被提升的对象的比例

Wilson and Moher [1989b], doi: 10.1145/74877.74882.

© 1989 Association for Computing Machinery, Inc., 经许可后转载

9.6.2 衰老半区

使用两个或者多个衰老半区对某一分代进行组织是实现延迟提升的方法之一。该策略要求对象在得到提升之前必须已经在其所属分代的来源空间与目标空间之间复制了多次。在 Lieberman 和 Hewitt 最初的分代回收器 [1983] 中,分代中的对象只有经历多次回收之后才可以集体提升。在图 9.2b 所示的衰老半区中,某一分代的存活对象是被复制到目标半区,还是被集体提升到下一个分代,取决于该分代中对象的整体年龄。该方案中,尽管较老的对象会有足够长的时间到达死亡,但最年轻的对象依然有可能过早地得到提升。

Sun 公司的 ExactVM[⊖]也将年轻代划分为两个半区(见图 9.2c),但其对年轻代对象的提升可以精细到单个对象级别,其方法是在对象头域中预留五个位来记录对象的年龄。回收器

⊖ 该系统后来更名为 Sun Microsystems Laboratories Virtual Machine for Research, 见 <http://research.sun.com/features/tenyears/volcd/papers/heller.htm>。

可以根据对象的年龄来判断是将其提升，还是将其复制到同一分代的目标半区中。该方案虽然可以避免提升最年轻的对象，但是其对年轻代存活对象的处理却会引入一些额外开销。

桶组 (bucket brigade) 和分阶系统 (step system) 可以更好地进行对象年龄鉴别，同时也无须为每个对象保留特定的空间以记录其年龄。该策略将每个分代内部划分为多个子空间，每次回收都会将一个桶或者阶中的存活对象递进式地复制到下一个，同时将最高阶中的存活对象提升到下一个分代。也就是说，在一个阶数为 n 的分阶系统中，对象在被提升到下一个分代之前必须经历过 n 轮回回收。Glasgow Haskell 允许将每个分代划分为任意多个阶 (默认的配置是年轻代包含两个阶，其他分代只有一个阶)，UMass GC Toolkit[Hudson 等, 1991] 也采用相同的策略。在 Shaw[1988] 的桶组策略中，每一阶又进一步被划分为两个半区，存活对象必须在阶内的两个桶之间经历过 b 次复制之后才可能被提升到下一阶，因此对于二阶桶组系统，对象在被提升到下一代之前必须经历 $2b-1 \sim 2b$ 次复制。Shaw 还对其策略进行调整以简化对象的提升。图 9.4 是其桶组策略的一个实例，其中 $b=3$ ，即对象在被移动到下一个衰老桶或者被提升到下一代之前，至少要经历 3 次复制。在 Shaw 的系统中，各分代在空间上是连续的，因此可以将衰老桶与年老代进行合并，即只有当最后一个桶的目标空间被填满时才进行存活对象的提升，提升的方法则是简单地调整两个分代之间的边界。图 9.2c 中的衰老空间 (aging space) 与此处的二阶桶策略存在一定的相似之处，但它却需要对存活对象头部中的年龄位进行额外的操作。

尽管“分阶”和“分代”策略都是根据年龄来划分对象，但不能将它们混淆。不同分代的回收频率不同，而同一分代内部的所有分阶则具有相同的回收频率。由于年老代的回收通常会比年轻代要晚，因而回收器必须记录从年老代指向年轻代的跨代指针，而跨阶指针则无需记录。将年轻代划分为多个分阶，不仅可以在无需为每个对象记录年龄的前提下避免过早的对象提升，而且还可以降低赋值器写屏障的开销。

9.6.3 存活对象空间与柔性提升

在上述所有基于半区复制的回收策略中，年轻代中一半的空间要用作复制保留区，因而其空间浪费率较高。为此，Ungar[1984] 进一步将年轻代划分成一个较大的诞生空间 (creation space) 以及两个较小的存活对象半区 (survivor semispace)，即桶 (见图 9.2d)。对象在诞生空间中分配，次级回收会将其中的存活对象提升到存活对象目标空间 (survivor tospace)。对于存活对象来源空间 (survivor fromspace) 中的对象，次级回收会根据其年龄决定是将其移动到年轻代内部的存活对象目标空间还是将其提升到下一代。诞生空间可

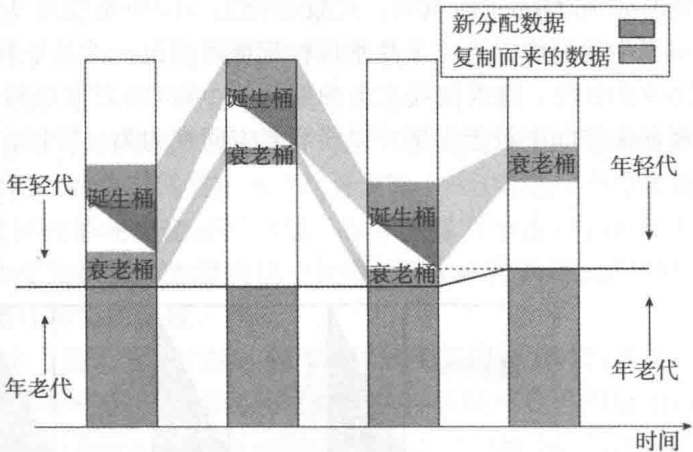


图 9.4 Shaw 的桶组系统。在年轻代内部，一次回收的存活对象会从其诞生空间复制到衰老半区。最后一个衰老半区与年老代相邻，因而回收器可以通过对两个分代边界的调整简单地实现对象提升

见 Jones [1996] 一书，经允可后转载

116
~
118

119

以远比两个存活对象半区大，因而这种空间组织方式可以提升空间利用率。例如在 Sun 的 HotSpot Java 虚拟机 [Sun Microsystems, 2006] 中，诞生空间与存活对象空间的大小比例是 32 : 1，即年轻代的复制保留区仅占用不到 3% 的空间^①。HotSpot 的对象提升策略并非要求对象达到一个固定的年龄，而是尝试为存活对象腾出一半的可用空间。相比之下，其他半区复制回收策略则会浪费一半的空间。

Opportunistic 垃圾回收器 [Wilson and Moher, 1989b] 使用桶组系统，同时配备一个较小的存活对象空间，该回收器中对象的提升年龄标准具有一定柔性。该回收器可以在不单独记录和操作每个对象年龄的前提下实现对象级别的提升控制。与其他策略相似，该回收器也将年轻代划分为一个诞生空间以及两个衰老空间，但衰老空间并非两个半区，而是采用分阶策略。次级回收将诞生空间中的存活对象移动到一个衰老空间中，同时将另一个衰老空间中的存活对象提升。如果仅依靠这一策略，则对象的提升标准为复制次数达到两次。但 Wilson 和 Moher 注意到，对象在诞生空间中是按照其分配的时间顺序排列的，因此只需要在诞生空间中引入一个高水位标记便可通过一次地址判断简单地区分出较为年轻的对象（即图 9.5 中位于高水位线之上的部分）。可以将诞生空间中较年轻的部分视为第 0 阶桶，同时将诞生空间中较老的部分以及衰老空间视为第 1 阶桶，只有第 1 阶桶中的存活对象才会得到提升。

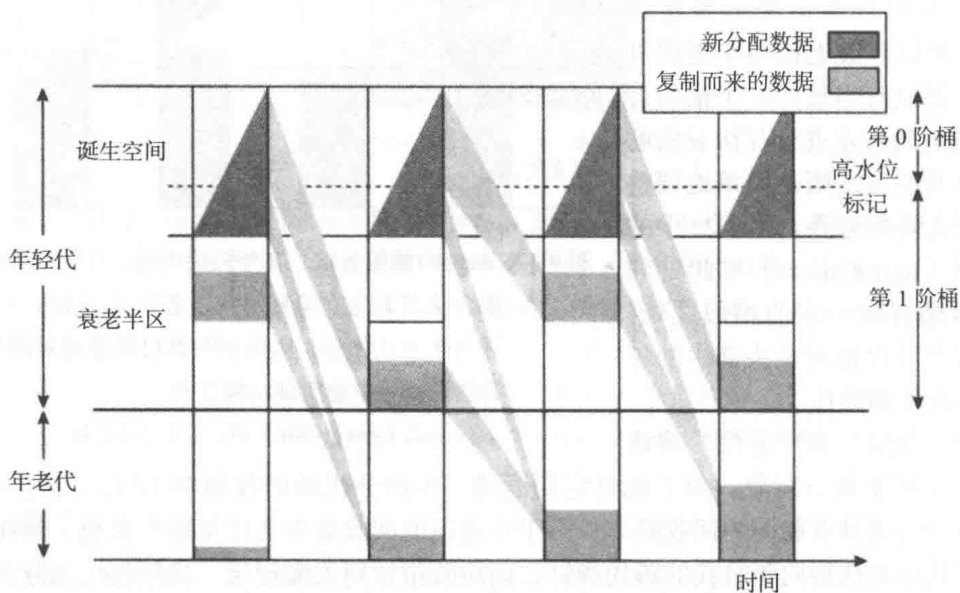


图 9.5 高水位标记。存活对象会从固定大小的诞生空间迁移到年轻代内部的衰老半区，并最终被提升到年老代。尽管回收器会将衰老半区中的所有存活对象提升，但借助于高水位标记，回收器通过一次地址比较便可简单地判定诞生空间中哪些对象应当提升

Wilson and Moher [1989b], doi: 10.1145/74877.74882.

©1989 Association for Computing Machinery, Inc., 经许可后转载

该策略将对象的提升年龄标准限定为最多经历两个次级回收，同时也不必显式记录对象

^① 硬件配置的发展速度很快。Ungar[1984] 的系统中，诞生空间为 140KB，存活空间为 28KB，年老代空间为 940KB。在 32 位的 Solaris 操作系统中，HotSpot 年轻代默认的大小是 2228KB。实践中我们使用过的极限配置是：3GB 的诞生空间，128KB 的存活空间，512MB 的年老代。

年龄或对其进行分区组织（例如我们前面所提到的半区组织方式）。诞生空间中，提升的阈值（即高水位标记——译者注）可以设定为 1 ~ 2 之间的任意小数，且可以在任意时间修改。图 9.3 展示了该算法的效果，其中的白色虚线代表高水位标记，其左侧深灰色区域以及黑色区域（即曲线 (c) 之下）中的所有对象都会在下次回收过程中得到提升。在高水位标记右侧，黑色区域中的对象将会得到提升，而灰色区域中的对象则会在下一次回收中死亡。Wilson 和 Moher 在字节码 Scheme-48 中使用了这一方案，且其使用了 3 个分代。标准 ML 也采用了该方案，但其分代数量增加到 14 个 [Reppy, 1993]。

9.7 对程序行为的适应

某些回收器可以在运行时根据程序的行为做出调整，例如 Opportunistic 回收器可以在程序运行时改变回收策略，其调整机制不仅可以达到很细的粒度，而且实现简单。真实应用程序（而非玩具式的程序或者人为的基准测试程序）的执行通常是分阶段的，同时对象生命周期的分布既不是随机的也不是静态的，因此回收器根据程序行为进行自适应调整是十分必要的。许多程序都具有一些共同的行为模式。某一集合中的对象可能逐渐积累，然后在同一时刻全部死亡。另外，存活对象的整体大小也可能在达到某一值后长期处于平稳。Ungar 和 Jackson[1988] 将对象成簇诞生但慢慢消亡的情况其类比为“蛇吞象”。一旦对象生命周期的分布模型不符合弱分代假说，则分代回收器便可能遇到问题。如果大量对象在存活到年老代之后才会死亡，则回收器的性能会受到影响。为解决这一问题，Ungar 和 Jackson[1988, 1992] 提出了多种柔性方案来控制年老代对象的数量。

垃圾回收器对赋值器行为的适应能力是十分有用的，例如可以减少期望停顿时间或者提升整体吞吐量。最简单的回收调度策略是仅当可用空间耗尽时才执行垃圾回收，但通用内存管理器可以通过对最年轻分代的大小进行调整来控制停顿时间：诞生空间越小，则次级回收所要提升的年轻代对象就越少。每个分代的空间大小也会影响对象的提升率。如果年轻代的空间太小以至于新生对象没有足够的时间到达死亡，提升率便会增大，而如果诞生空间很大，则回收时间间隔会增大，对象的提升率也会降低。

9.7.1 Appel 式垃圾回收

Appel[1989a] 针对标准 ML 提出了一种自适应分代策略，其年轻代可以在堆内存总量不变的情况下占用尽可能多的空间（而不是一个大小固定的空间）。ML 语言中，一次回收完成后通常只有不到 2% 的对象可以存活，而该策略正是针对这一情况而设计的。Appel 的方案将堆划分为 3 个区域：年老代 (old)、复制保留区 (copy reserve)、年轻代 (young) (见图 9.6a)。针对年轻代的回收会将其中所有存活对象集体提升到年老代的末尾 (见图 9.6b)。回收完成后，年老代之外的其他空间将被划分为大小相同的两份，一份用作复制保留区，另一份则作为年轻代的诞生空间。如果年轻代的可用空间小于某一阈值，则会执行整堆回收。

[121]

与其他基于复制的回收策略一样，Appel 式垃圾回收必须保证复制保留区在最差情况下仍可以容纳所有的存活对象（即年轻代和年老代存活对象的总和）。最保守的策略是确保 $old + young \leq reserve$ ，但 Appel 却可以在降低整堆回收频率的同时将这一要求降低为 $old \leq reserve$ 且 $young \leq reserve$ ，其证明如下。在次级回收之前，即使所有的年轻代对象都存活下来，复制保留区也能够将其全部容纳。次级回收完成后，所有刚刚完成提升的对象均位于“old”所标识的区域，由于它们都是存活对象，因而即使立刻执行整堆回收，这些对象也

无需移动。同时由于 $old \leq reserve$ ，新的复制保留区也可容纳 old 所标识区域中所有在以往的次级回收中得到提升的对象（见图 9.6c）。主回收完成后，回收器需要将 old 区域中所有的存活对象（当前是在堆的顶端）移动到堆的底部[⊖]。需要注意的是，对于一部分在年轻代而另一部分在年老代的环状垃圾，这种“二次回收”的方法无能为力，但在下次主回收中，这种环状垃圾必然全部位于年老代，因此最终可以得到回收。

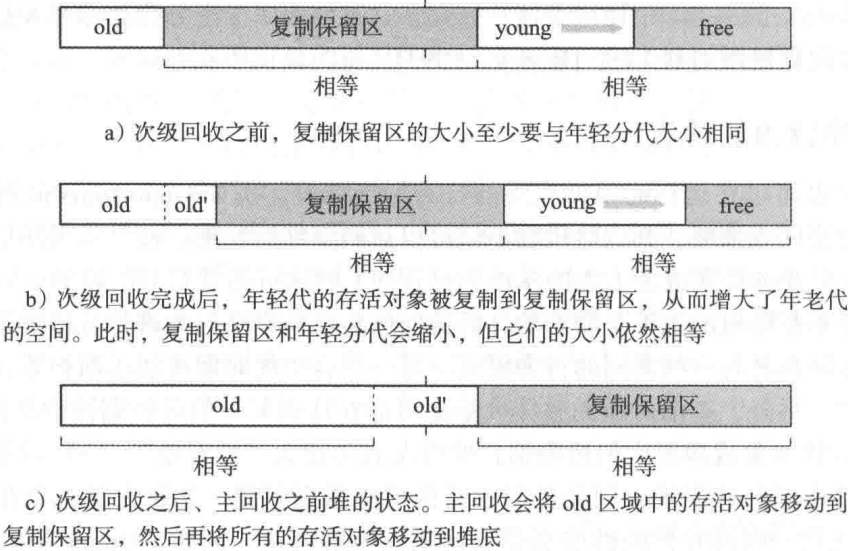


图 9.6 Appel 的简单分代回收器（灰色区域为可用空间）

Appel 的分代回收器基于连续的堆空间，但也有其他 Appel 式的回收器使用块结构堆，后者可以避免在主回收完成后将存活对象移动到堆底这一操作。在年轻代具有收缩能力的基础上，年老代也可使用诸如标记 - 清扫等非移动式回收算法来管理。

与使用集体提升策略且年轻代大小固定的分代回收器相比，Appel 式回收器的优点在于其复制保留区的大小可以动态调整，从而可以提升内存利用率并降低回收频率。但是，为避免回收器出现性能颠簸，仍有一些细节需要注意。Appel 的设计出发点在于许多程序都具有较高的分配率以及较小的提升率。如果诞生空间收缩得太小，则次级回收发生的频率会变高，但由于得到提升的对象太少并不足以触发主回收，因而程序的性能会受到影响。解决这一问题的方法是在年轻代空间小于某一阈值时便进行整堆回收。

9.7.2 基于反馈的对象提升

还有许多控制提升率的策略均以降低停顿时间为目标。基于反馈的统计分析提升 (demographic feedback-mediated tenuring) [Ungar and Jackson, 1988, 1992] 尝试对刚提升不久便立即死亡的对象进行控制以减缓停顿时间过长的情况。该方案使用一次回收所提升对象的总量来预测下次回收将要提升的对象总量，并据此减少或者增大提升率。如果对象存活率超过某一最大值，则在下次回收过程中，判定对象是否应当提升的年龄标准会增大。尽管这一策略可以控制对象的提升率，但它却无法将年老代对象降级到年轻代。Barrett 和 Zorn[1995] 在不同分代之间引入具有双向移动能力的危险边界 (threatening boundary)，但由于回收器无法预测分代之间的边界未来会落在哪里，因而写屏障需要对更多的指针进行追踪。

⊖ 即将 old' 和复制保留区中所有存活的年老代对象移动到最左端。——译者注

122

Sun 的 HotSpot 回收器家族在 1.5.0 版本中引入了 Ergonomics 机制, 其目的在于根据不同的用户需求来调整各分代的大小。Ergonomics 的设计目标并不是满足硬实时要求, 而是以 3 个软目标作为出发点的。Ergonomics 首先尝试满足最大停顿时间的要求, 然后在此基础上尝试提升吞吐量 (通过测量垃圾回收占程序整体执行时间的比例来判定), 最后在前两个目标都达到的前提下尽量减少程序所占用的空间。优化停顿时间的方法是: 对每次回收的停顿时间进行统计分析, 找出时间消耗最长的分代, 然后缩小该分代所占据的空间。吞吐量的提升则是通过增大整个堆或者某个分代空间来实现的, 但分代空间的增大同时也会导致该分代的回收时间变长。默认情况下, 堆空间通常更倾向于增大而不是收缩。

Vengerov[2009] 提出了一种分析 HotSpot 回收器吞吐量的模型, 并基于该模型得出一种实用的回收器调整算法。该算法主要关注 HotSpot 回收器两个分代的相对大小、对象提升阈值、年轻代在得到提升之前所需经历的回收次数, 并在运行时对其进行动态调整。Vengerov 经过观察得出一个重要结论, 即对提升率阈值的调节不能仅考虑这一操作所能减少的对象提升数量, 还要考虑主回收完成之后年老代的可用空间与每个次级回收所提升对象总量的比值。Vengerov 在其 ThruMax 算法中提供了一种可以交替调整年轻代空间大小以及提升率的协同演化框架, 其工作流程大致如下: 在 HotSpot 回收器的第一次主回收之后, 或者存活对象的总量达到某一稳定状态之后 (即连续两个次级回收之间年轻代存活对象总量的 75% ~ 90%), ThruMax 逐渐增大诞生空间的大小 S , 直至其到达一个临近的最佳值 (即 S 开始下降, 或者在某个值附近摆动), 然后 ThruMax 对提升阈值进行调整, 直到吞吐量出现下降趋势为止。如此一来, 回收器既可以在避免任何副作用的前提下适当减少诞生空间的大小 S , 也能确保在进行足够多的次级回收之后才需要进行主回收。

总之, 像 HotSpot 这样的复杂回收器中可能会存在大量的可调参数, 但每个参数之间可能存在一定的依赖关系。

9.8 分代间指针

在对某个分代进行回收之前, 回收器必须先确定该分代的根。正如我们在图 9.1 中所看到的, 某个分代的根不仅包括寄存器、栈、全局变量中的指针值, 如果该分代内部的对象被堆中其他空间的对象所引用, 且这些空间不会与该分代同时进行回收, 则这些引用也属于该分代的根。这些引用通常来自于更老分代, 或者各分代之外的堆空间, 例如大对象空间, 或者永远不会进行回收的空间 (如永生对象或代码所占据的空间)。分代间指针的创建有 3 种方式: 一是在对象创建时写入, 二是在赋值器更新指针槽时写入, 三是在将对象移动到其他分代时产生。回收器必须要对分代间指针进行记录, 只有这样才能确保在对某一分代单独进行回收时根的完整性。我们将所有需要记录的指针统称为回收相关指针 (interesting pointer)。

另一种需要关注的引用来源是存在于堆之外的引导映像 (boot image) 中的对象 (即引导对象), 这些对象自从程序启动之后便一直存在。通用垃圾回收系统至少可以使用 3 种方式来处理这些对象: 第一, 对引导对象进行追踪 (trace), 其优势在于如果某一堆中对象仅从某一引导对象可达, 而该引导映像对象本身不可达时, 回收器可以将该堆中对象回收; 第二, 对引导对象进行扫描, 并从中找出指向我们所关注的分代的指针; 第三, 对引导映像对象中的回收相关指针进行记录。追踪的代价较高, 通常只会在整堆回收时使用, 因此追踪通常与扫描或记忆集结合使用。扫描的优势在于当赋值器对引导映像对象进行更新时无需使用

额外的写屏障，但其缺陷在于回收器必须搜索更多的域才能找全回收相关指针。如果将扫描与追踪相结合，则回收器在追踪完成后必须将不可达引导映像对象的各指针域清零，否则将错误地导致某些垃圾对象重新可达。记忆集同样也拥有其自身的优势与开销，也不需要将不可达引导映像对象的指针域清零。

9.8.1 记忆集

记忆集^①是用于记录分代间指针的数据结构，其中所记录的是从堆中一个空间指向另一个空间的指针来源（如图 9.1 中的对象 U 以及对象 S 的第二个槽）。这里需要注意的是，记忆集中所记录的是回收相关指针的来源而非目标，其原因有二：第一，在移动式回收器中，一旦目标对象被复制或者提升，回收器便可根据记忆集更新回收相关指针的来源。第二，在连续两次回收过程之间，某个回收相关指针的来源域可能会多次被修改，如果记录回收相关指针的来源而非目标，则回收器可以仅对回收时刻该指针域所引用的对象进行处理，从而无需考虑其曾经指向过哪些其他对象。因此，每个分代的记忆集均只需记录可能指向该分代内部对象的回收相关指针来源。不同的记忆集实现方式在来源地址的记录方面所能达到的精度也各不相同。精度并非越高越好，较高的精度通常会增大赋值器的额外开销、记忆集空间开销以及回收器处理记忆集的时间开销。需要注意的是，此处记忆集中的“集”并非严格意义上的集合，其具体实现中通常允许出现元素的重复，此时记忆集便成为“多集合”(multiset)。

需要探测和记录的指针当然是越少越好。回收器所执行的指针写操作（例如移动对象）通常很容易探测到。赋值器所执行的指针写操作可以用软件写屏障的方式实现，即编译器在每个指针写操作之前插入额外的指令，但如果缺乏编译器的支持，该方案的实现通常会遇到问题。这种情况下，通常需要借助于操作系统的虚拟内存管理器来获取写操作发生的地址。

在不同的编程语言及其实现中，指针写操作在全部赋值器操作中所占的比例各不相同。Zorn[1990]通过对一套 SPUR Lisp 程序进行静态分析发现，指针写操作的比例为 13% ~ 15%，而 Appel 对 Lisp 进行静态分析则将这一数字降低到 3%[1987]，对 ML 进行动态分析的结果则是 1%[1989a]。在基于状态的语言中，破坏性（destructive）指针写操作出现的比例通常更高。Java 应用程序中，指针写操作的比例通常变化较大，如 Dieckmann 和 Hölzle[1999]发现在所有的堆访问操作中，指针写操作的比例为 6% ~ 70%（此处的最大值应该是一个异常值，次大的值是 46%）。

9.8.2 指针方向

并非所有的写操作都需要进行回收相关指针的探测和记录。某些语言（例如 ML 的某些实现）将程序的活动记录置于堆中，如果每次回收都需要把这些帧当作根集合来扫描，则帧中所包含的指针域可以用我们在 11 章中将要介绍的技术来查找。此时如果编译器可以检测出针对栈槽的写操作，则可以免去该过程中的写屏障开销。另外，许多写操作所涉及的对象都是在同一个分区内，尽管此类写操作可能会被探测到，但它不会产生任何回收相关指针，因此也无需记录其来源。

如果我们对各分代进行回收的顺序施加一定的限制，则需要记录的分代间指针的数量还可以大幅降低。如果可以确保在对年老代进行回收的同时一定会回收年轻代，则无需记录从年轻代到年老代的指针（例如图 9.1 中对象 N 所包含的指针）。许多指针写操作都是在对新

① 我们此处的术语与 Jones[1996]有所不同，后者将卡表（card table）与其他记忆集实现方案区别对待。

创建对象进行初始化时发生的, Zorn[1990] 估计 Lisp 语言中 90% ~ 95% 的指针写操作都发生在对象初始化过程中(在剩余的写操作中, 2/3 的写操作都发生在年轻代内部)。从概念上讲, 新创建对象中的指针必然指向年龄更大的对象, 但不幸的是, 在许多语言中, 对象的分配与其内部域的初始化是两个相互独立的过程, 这导致编译器无法将初始化过程与非初始化过程相区分, 而后者极有可能创建从年老代指向年轻代的引用。某些语言中, 编译器拥有更强的指针写操作判断能力, 从而无需引入写屏障。例如, 在诸如 Haskell 的懒惰式纯函数式语言中, 大多数指针写操作都会指向更老的对象, 而只有真正对一个待计算值(thunk, 即已经绑定了参数的函数)进行计算并用一个指针将其覆盖时, 才有可能创建从年老代指向年轻代的指针。对于 ML 这种有副作用的严格编程语言, 开发者必须对可写变量进行显式声明, 而只有对这些对象进行写操作, 才可能创建从年老代指向年轻代的指针。

对于诸如 Java 之类的面向对象语言, 情况则稍为复杂。面向对象语言的编程范式通常集中在对象状态的更新上, 这自然会导致从年老代指向年轻代的指针出现得更频繁。然而, 大多开发者都会编写函数式风格的代码, 这通常可以避免副作用, 同时许多应用程序中绝大多数指针的写操作均是针对年轻代对象的初始化。但 Blackburn 等 [2006a] 指出, 不仅是在不同应用程序之间, 同一程序内的不同个体之间也存在着相当大的行为差异。从他们的报告中我们可以看出, 指针写操作会在指针的方向以及距离(此处是指来源对象与目标对象在创建时间上的距离)上表现出较大的差异性, 其原因之一便是可能存在许多针对同一区域的写操作, 这对记忆集的实现有着重要的影响。

如果在任何情况下都对年轻代进行整体回收, 则写屏障可以忽略对新生区的写操作(可以预期, 此类操作通常较为普遍)。但如果堆中具有多个独立的回收区域, 则可能需要使用不同的指针过滤器。例如, 回收器可能会使用启发式方法来估算哪个区域的存活对象最少, 进而确定对不同区域进行回收的优先级。这种情况下, 我们必须对两个方向上的指针都进行记录, 因而需要记录的指针数量通常较多, 此时的最佳选择是选用整体空间大小与内部所记录指针数量无关的记忆集。我们将在第 11 章讨论写屏障和记忆集的具体实现。

[125]

9.9 空间管理

年轻代存活对象的归宿有两个, 一是被复制到同一分代的其他半区, 二是被提升到年老代。基于年轻代存活对象较少这一假设, 我们通常期望增多且简化年轻代的回收。一旦要对年老代进行回收, 通常也需同时回收所有年轻代, 否则就需要在写屏障中对双向指针都进行记录。通常对最老代的回收会触及堆中的所有其他区域, 但永生对象区域或者引导映像区除外, 这些区域中的引用通常会被看作根, 且回收器也需要更新其指针域。整堆回收无需使用记忆集(永生对象区域以及引导映像区除外, 但如果整堆回收也会扫描这些区域, 则记忆集便彻底不再需要了)。

有许多策略可以用于最老代的管理。半区复制策略是一种可选方案, 但它可能并非最佳选择。半区复制需要在堆中开辟一块复制保留区, 这会导致堆可用空间的减少, 从而增大了各级回收的频率。半区复制同时也可能导致长寿对象在两个半区之间来回复制。标记-清扫回收器的内存使用率较高, 特别是在堆空间较小的情况下 [Blackburn 等, 2004a]。尽管标记-清扫回收所使用的空闲链表分配通常比顺序分配要慢, 且其局部性无法预测, 但这通常仅在对象分配较为频繁的年轻代时才会成为问题 [Blackburn 等, 2004a]。标记-清扫回收的主要缺陷在于它是非移动式回收, 从而有可能加剧年老代的内存碎片情况, 针对这一问题可

以引入额外的整理式回收，即在年老代碎片率达到一定程度时进行整理（而非每次回收都进行整理）。整理式回收也可以较好地处理长寿对象。正如我们在第 3 章所提到的，整理式回收通常会在年老代的底部形成一个“密集前缀”（dense prefix）。HotSpot 标记 - 整理式回收器只有在这一“沉积区”的碎片率达到一定程度（可以由用户决定的）时才会进行整理。

分代垃圾回收器通常会在物理上隔离不同的分代，这便要求年轻代使用复制式回收进行管理。Appel 式回收器等较为保守的回收器要求复制保留区在最差情况下都能够容纳所有存活对象，但在实践中，一次回收后年轻代的存活对象通常较少。

如果使用较少的复制保留区，并且可以在保留区空间不足的情况下切换到整理式回收，则空间利用率可以大幅提升 [McGachey and Hosking, 2006]。但是，由于复制保留区空间不足的情况只有在回收过程中才会发生，所以回收器必须能够随时在复制和标记两种操作之间进行切换。图 9.7a 所展示的是回收器完成所有存活对象鉴别之后堆的状态，此时复制保留区已被填满，其中已完成复制的对象为黑色，其余的年轻代存活对象为灰色。接下来的操作将是把所有已标记对象整理到新生代的一端（见图 9.7b），这通常需要多次遍历。不幸的是，整理过程会破坏年轻代黑色对象中所记录的转发地址。McGachey 和 Hosking 的解决方案是：先对灰色对象进行一次遍历并更新其中指向黑色对象的引用，然后再使用 Jonkers 的滑动式整理器（参见 3.3 节）移动已标记的灰色对象，这一引线算法无需在对象头域中占用额外的头域。更好的解决方案可能是使用 Compressor 算法（参见 3.4 节），因为它不仅无需引入任何额外头域，而且也不会修改存活对象的任何一个域。他们基于 MMTk 回收器进行实验并发现，如果将复制保留区的大小设置为堆空间的 10%，则无论年老代使用复制式还是标记 - 清扫回收，整体性能平均都会提升 4%，某些情况下甚至会达到 20%。

126

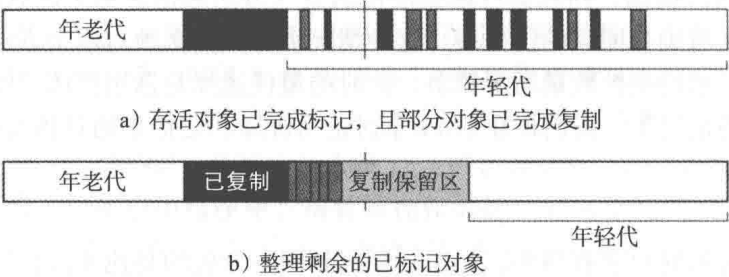


图 9.7 年轻代在复制和标记之间进行切换：a) 复制保留区已满。年轻代中的黑色对象已经复制到年老代，灰色对象已完成标记但尚未复制，其他对象均为死亡对象；b) 整理过程将灰色对象滑动到紧邻年老代的一端，并扩展年老代的大小

9.10 中年优先回收

事实证明，对于许多应用程序而言，分代垃圾回收都可以十分高效地管理寿命较短的对象。但正如我们在 9.7 节所看到的，寿命稍长[⊖]的对象处理起来则较为棘手。分代垃圾回收本质上是仅回收堆中较为年轻的对象，同时忽略其他较老对象，但在每次回收中，年轻对象集合的具体范围取决于需要回收的分代的数量，即：是仅回收新生代，还是需要回收中间分代（在使用超过两个分代的系统中），或者是要进行整堆回收。某些自适应策略具有控制对象提升率的能力，我们可以将其看作是通过对各年轻代的年龄范围进行调整，使得年轻代对象

⊖ 但不是最长。——译者注

有更多的时间到达死亡。但分代垃圾回收并非避免整堆回收的唯一途径（我们将在下一章介绍其他基于非年龄特征的回收策略），基于年龄的回收（age-based collection）策略可能包括：

仅针对最年轻分代的回收（分代回收）（youngest-only collection）：回收器仅处理堆中最年轻的对象。

仅针对最年老分代的回收（oldest-only collection）：可以想象一个仅处理堆中最年老对象的回收器，即假定更老的对象更容易死亡。但这一策略通常都十分低效，因为它会花费大量的时间频繁地处理永生对象或者非常长寿的对象。而正是这一原因，使许多回收器刻意避免对这些古老的“沉积物”进行处理。

中年优先回收（older-first collection）：回收器将主要精力集中在中年（middle-aged）对象的处理上。这一策略可以确保年轻对象有足够的时间到达死亡，同时也避免对长寿对象进行处理（当然也会偶尔对其进行处理）。

127

中年优先回收存在两个技术难点：一是如何确定中年对象，二是由于两个方向的回收相关指针（从年老到中年、从年轻到中年）都需要记录，因而记忆集的管理复杂度会有所提升。下面我们将介绍两种解决方案。

更新中年优先回收（renewal older-first garbage collection）。一种获取对象“年龄”的方案是计算对象自创建以来所经历的时间，或者距其上一次经历垃圾回收的时间，并以两者的最小值为准 [Clinger and Hansen, 1997; Haansen, 2000; Hansen and Clinger, 2002]。更新中年优先回收通常仅回收堆中“最老”的前缀。为简化记忆集的管理，回收器将堆划分为 k 个大小相等的阶，同时将编号最小的空阶用于对象分配。当堆空间耗尽时，回收器将对最老的 $k \sim j$ 阶进行回收（见图 9.8 中的灰色窗口），并将其中的存活对象复制到与第 1 阶相邻的复制保留区（图 9.8 中的黑色区域）。此时，存活对象将得到“更新”，此时第 $j \sim 1$ 阶成为最老的阶。在图 9.8 中，堆在虚拟地址上向右增长，从而简化了写屏障的设计，即记忆集只需要对方向从右向左且来源地址大于 j 的指针进行记录。这种堆排列方式在 64 位的应用程序中通常不会遇到问题，但是在 32 位系统中则很容易耗尽虚拟地址空间。为此，更新中年优先回收必须在触达堆的末端时回绕到堆的始端，并且对各阶重新进行编号，此时写屏障在捕捉回收相关指针时便不能简单地依赖地址比较，它必须对来源和目标对象所在阶的编号进行比较。该方案的第二个缺点在于，对象在堆中的排列顺序并非按照其真正的年龄进行的，而是不可逆地混杂在一起。Hansen 在 Scheme 语言的 Larceny 实现中引入一个标准的新生代以过滤掉大量回收相关指针（然后仅使用更新中年优先回收方式管理年老代），但其记忆集所占用的空间依然十分庞大。

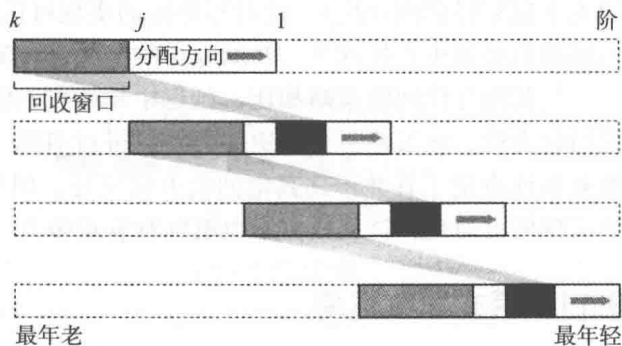


图 9.8 更新中年优先回收。每次回收完成后，存活对象将紧挨最年轻的对象存放

延迟中年优先回收（deferred older-first garbage collection）。该方案可以确保对象在堆中按照其真实年龄排列 [Stefanović 等, 1999]。延迟中年优先回收使用一个固定大小的回收窗口（图 9.9 中的灰色区域），并在堆中沿着年老到年轻的方向进行滑动。当堆空间耗尽时，回收器仅对回收窗口内的对象进行回收，同时忽略其他更老的或者更年轻的对象

128

(图 9.9 中的白色区域)。回收完成后,所有存活对象(图 9.9 中的黑色部分)都被将复制到最老对象之后的区域,而回收所得的空间将被添加到堆的最年轻端(即最右端)。下一轮的回收窗口将与存活对象的年轻端紧邻。该方案寄希望于回收器可以在堆中达到一个最佳回收状态,在该状态下,回收窗口中仅有少量对象存活,同时回收器也拥有较低的标记-构造率,因而回收窗口将会以十分缓慢的速度滑动(正如图 9.9 中最后一行所示)。然而,有时,当回收窗口触达堆的最年轻边界时,回收器便需要将回收窗口重置到堆的最老端。在延迟中年优先回收算法中,尽管对象依照其真实年龄排列,但其赋值器写屏障的实现却十分复杂,因为其必须对所有从年老区指向回收窗口的指针、所有老-新指针、所有的新-老指针(除非该指针的来源位于回收窗口中)进行记录。类似的,回收器的复制写屏障也必须记录所有从存活对象指向其他区域的指针、所有从年轻存活对象指向年老存活对象的指针。另外,延迟中年优先回收通常会将堆划分为多个内存块,每个内存块都有其自身的“死亡时间”(且必须确保较老内存块的死亡时间大于较年轻的内存块)。此时写屏障的实现可以以内存块死亡时间的比较结果为基础进行,但同时也必须小心处理死亡时间溢出的情况 [Stefanović 等, 2002]。

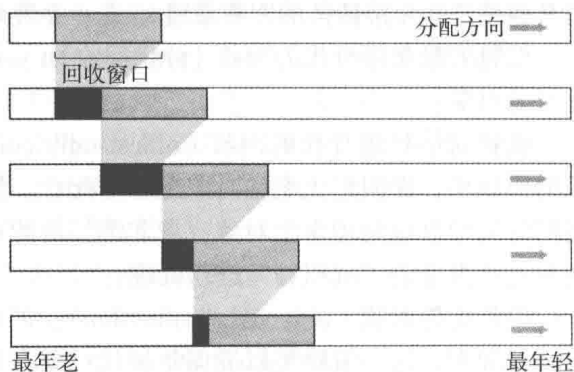


图 9.9 延迟中年优先回收。该算法使用一个中年回收窗口来选定需要回收的对象。一次回收完成后,存活对象将紧邻上一次回收的存活对象存放。该算法期待回收器找到一个存活率很低的最佳状态,此时回收窗口将以很慢的速度滑动

与其他分代回收策略相比,延迟中年优先回收可以降低最大停顿时间,但与更新中年优先回收类似,该策略需要对更多的指针进行追踪。在地址空间较小时,中年优先回收写屏障的复杂性决定了其并不比其他回收方案更好,但当地址空间更大时(如 64 位系统),写屏障的实现便可简化,该回收方案也更加具有竞争力。

129

9.11 带式回收框架

本章我们已经介绍了多种不同的基于年龄的回收策略,这些回收策略的基本指导思想大体上可以总结如下:

- “大多数对象都在年轻时死亡”,即弱分代假说 [Ungar, 1984]。
- 分代回收器会避免对年老对象进行频繁回收。
- 增量回收可以改善回收停顿时间。在分代垃圾回收器中,新生代的空间通常较小;其他回收技术通常也会限制待回收空间的大小,例如成熟对象空间回收器 (mature object space collector) (也称为火车回收器) [Hudson and Moss, 1992]。
- 在较小的诞生空间中使用顺序分配可以提升数据的局部性 [Blackburn 等, 2004a]。
- 对象需要足够的时间到达死亡。

带式垃圾回收框架 (beltway garbage collection framework) [Blackburn 等, 2002] 对上述所有思想进行了综合,它可以配置成任意一种基于分区策略的复制式回收器。带式回收器的一个回收单元称为回收增量 (increment),多个回收增量可以组合成队列,称为回收带 (belt)。图 9.10 中,每一行代表一个回收带,回收带上的每个“托盘”代表一个回收增量。

尽管每次回收所选定的回收增量通常是最年轻回收带中非空且最老的一个，但整体来说，回收带内部的各回收增量通常依照先进先出的方式进行独立回收，各条回收带之间同样也符合先进先出的回收顺序[⊖]。提升策略决定了一次回收完成后存活对象的目标空间，即它们有可能被复制到同一回收带中的其他回收增量里，也有可能被提升到更高级回收带中的某一回收增量里。需要注意的是，带式回收器并非另一种分代回收器，更不能简单地将回收带与分代混为一谈。其主要区别在于，分代回收器通常会回收某一回收带中的所有回收增量，而带式回收框架可以对每个回收增量独立进行回收。

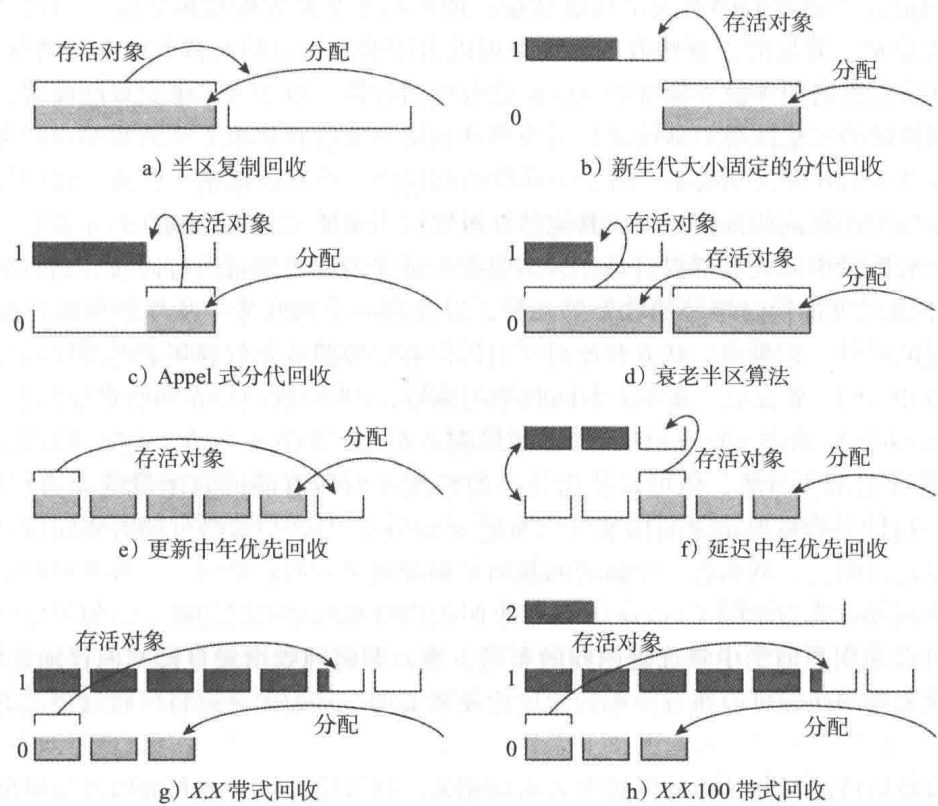


图 9.10 带式回收框架。该框架可以配置成任意一种复制式回收器。每一幅图演示了其对应算法中新生对象将在哪个回收增量中创建、哪个回收增量会优先得到回收、存活对象将被复制到哪个回收增量中

Blackburn et al [2002], doi: 10.1145/512529.512548.

©2002 Association for Computing Machinery, Inc., 经许可后转载

图 9.10 展示了带式回收器对其他回收算法的模拟，其中的某些算法我们已经在前面的章节中介绍过，而另一些则是在本书中新出现的回收算法。简单的半区复制回收器可以看作一个包含两个回收增量的回收带（见图 9.10a），每个回收增量即为堆空间的一半。一次回收完成后，第一个回收增量（即来源空间）中的所有存活对象被复制到第二个回收增量（即目标空间）中。分代垃圾回收器的每个分代都可以看作是一个独立的回收带。对于新生代大小固定的分代回收器而言，其第 0 个回收带内部的回收增量无法变化（见图 9.10b），而在 Appel 式回收器中，两个回收带中的回收增量都可以自由增长直至堆可用空间耗尽（见图

⊖ 这里的先进先出的意思是越早用于分配的回收带将越早得到回收。——译者注

9.10c)。基于衰老半区的回收算法可以通过增大第0个回收带中回收增量的数量来模拟(见图9.10d)。但与9.6节所介绍的衰老半区算法不同,此处增加回收增量的目的是减少停顿时间,因为次级回收并不会处理第二个回收增量中的不可达对象。带式回收框架也可以模拟更新中年优先回收与延迟中年优先回收。图9.10e可以清晰地反映出更新中年优先回收是如何将不同年龄的对象混杂在一起的。延迟中年优先回收使用两个回收带,当回收窗口触达第一个回收带的最年轻端时,回收器需要将两个回收带置换(见图9.10f)。借助于带式回收框架,Blackburn等人还开发出了其他一些新的复制式回收算法。 XX 带式回收算法(见图9.10g)相当于是在Appel式回收器中引入了回收增量,即当第1个回收带被填满时,回收器仅处理第一个回收增量。此处的 X 意味着回收增量可以占用的最大空间占整个堆空间的百分比,因此100.100带式算法即等价于标准的Appel式分代回收器。对于 XX 带式算法而言,如果 $X < 100$,则算法的完整性得不到保证,因为环状垃圾可能会跨越第1个回收带中的多个回收增量。而在 $XX.100$ 带式算法中,第3个回收带仅包含一个回收增量,且该回收增量可以增长到与堆空间大小相同的规模,因而其完整性可以得到保证(见图9.10h)。

130

假设在所有配置中回收器都仅对最年轻回收带中最老的回收增量进行回收,则写屏障仅需要记录从年老回收带指向年轻回收带的指针,以及同一个回收带中从年轻回收增量指向年老回收增量的指针。如果我们从0开始对所有的回收带按照从年轻到年老的顺序编号,则每个增量可以用 $\langle b, i \rangle$ 来表示,其中 b 为回收带的编号, i 为回收增量在回收带中的索引号。此时,对于从 $\langle b_i, i \rangle$ 指向 $\langle b_j, j \rangle$ 的指针,如果满足 $b_j < b_i \vee (b_j = b_i \wedge i < j)$,则写屏障需要对该指针进行记录。当然,也可以使用较小的整数 n_i 对所有的回收增量统一进行编号,此时判定某一指针是否需要记录的标准可以简化为 $n_j < n_i$,但这一策略可能需要偶尔对各回收增量的编号进行调整,例如将一个新的回收增量添加到某一回收带时。一种典型的实现方案是将地址空间划分为多个帧(frame),而每个回收增量都位于不同的帧上。如果地址空间足够大,则可以采用类似于中年优先回收的布局方案,即将回收增量直接与内存地址绑定,此时新老回收增量的比较可以通过简单的地址比较来实现,从而无需再将回收增量映射到具体的编号。

131

带式回收器的性能与其具体的配置方式密切相关。回收带在堆中的布局以及写屏障的实现至关重要,因为这不仅决定了哪些指针需要记录,而且决定了哪些对象需要复制,以及复制的目标空间。

9.12 启发式方法在分代垃圾回收中的应用

分代垃圾回收器可以较好地处理短命对象,但其对长寿对象的处理能力则略显不足,这一问题主要表现在两个方面:第一,年老代垃圾的回收不够及时,因为没有哪种策略可以确保在年老代出现大量垃圾时尽快将其回收;第二,年老代的所有长寿对象都必须是从年轻代复制而来的,同时为避免过早地提升对象,某些回收器要求年轻代对象在得到提升之前必须在年轻代中经历数次回收。对于长寿对象而言,这些复制操作都相当于是无用功,更好的解决方案是直接将长寿对象预分配到你最终可能到达的分代中。

一些研究者尝试通过分析程序特定代码位置所分配对象的生命周期分布来解决这一问题。这一方案对于虚拟机的开发者来说具有一定的可行性,因为他们可以知道在其具体的虚拟机中哪些数据结构会是永久性的、哪些库或者代码对象永远不会或者至少不太可能被卸载。这些对象的预分配逻辑可以在虚拟机内部实现。

也有研究者通过动态分析 (profiling) 的方法识别长寿对象。Cheng 等 [1998] 尝试记录程序中哪些位置所分配的对象始终会得到提升。Blackburn 等 [2001, 2007] 统计堆达到最大规模时程序不同位置所分配的存活对象在堆中的比例, 并以此作为寿命指标来预判新分配对象可能的生命周期。这两种技术都需要对程序进行离线追踪并进行汇总分析。开发者可以根据这些信息对代码进行优化, 即将新生对象预分配到最合适的一个分代或永久对象空间中。该方案的不足之处在于, 每种预提升方案可能都只适用于某一特定的程序, 为此, Blackburn 等人提供了一种通用的解决方案来判定程序中哪些位置所分配的对象需要进行预分配 (以引导映像或者代码库的形式加载到程序中)。这一通用解决方案的有效性进一步说明了对程序进行离线分析的必要性。

Marion 等 [2007] 提出了一种能够实现真正预测 (而非自我预测) 的通用方案, 该方案通过对程序微观范式 (micro-pattern) [Gil and Maman, 2005] 进行句法比较 (syntactic comparison) 来判定哪些对象需要预分配。该方案的实现需要依赖一个事先生成的知识库 (通过机器学习方法对大量程序的追踪结果进行分析, 进而统计出每种微观范式所分配对象的生命周期特征)。Harris [2000] 和 Jump 等 [2004] 使用实时采样的方法进行预分配判定, 并取得一定的性能提升。此类方法通常可以较好地判定出程序中生命周期趋向于永久的对象的分配, 但其对于简单的长寿对象或者中等寿命对象的判定能力则稍显不足。

基于“相关对象通常具有相似的生命周期”这一规律, Guyer 和 McKinley [2004] 尝试将相关对象放置在一起。他们使用编译时静态分析的方法来判定新分配对象将与哪个已分配对象相关, 然后将新生对象与其相关对象放置在相同的空间。这一分析过程既不需要执行得十分彻底, 也不强制要求将具有相似生命周期的对象放置在一起。该策略不仅可以减少复制次数、显著降低回收时间, 同时还可以减轻写屏障的压力。

对于懒惰函数式语言中的分代垃圾回收器, 只有更新待计算值 (thunk) 时才需要引入写屏障, 因为其他的写操作必然都是针对较为年轻的对象。每个待计算值最多只会更新一次, 除此之外其他对象都不可改变。Marlow 等 [2008] 将这一观察结果应用在基于阶 (step-based) 的分代式回收器中, 即尽量将对象提升到与其某一引用来源相同的分代或者阶, 理想情况下, 这一引用来源应当是该对象的所有引用来源中最老的一个。即使对于可写对象, 所有针对新生对象的写操作也必然不会导致回收相关指针的出现。Zee 和 Rinard [2002] 使用静态分析的方法来去除 Java 语言中针对这些对象的写屏障, 其结果表明, 某些应用程序的整体执行时间会因此得到小幅提升。

132

9.13 需要考虑的问题

事实证明, 分代垃圾回收器提供了一种高效的对象组织方式, 并且可以显著提升许多应用程序的性能。如果限制最年轻分代的整体大小, 且将回收的主要工作集中在这一代, 则许多应用程序的期望停顿时间可以降低到难以察觉的水平。分代的组织方式同样也可以提升程序的整体吞吐量, 这表现在两个方面: 第一, 长寿对象的处理频率降低, 这不仅减少了长寿对象的处理开销, 而且使得中年对象有足够的时间到达死亡 (因此无需对其进行追踪); 第二, 分代回收器在分配新生对象时通常使用顺序分配的策略, 其内存访问模式的可预测性提升了程序的局部性, 同时由于大多数写操作都是针对最年轻对象发生的, 赋值器的局部性得到了进一步提升。

但是, 分代垃圾回收器并非一剂万能的灵丹妙药, 其性能表现严重依赖于应用程序中对

象的生命周期分布。分代式回收器对年轻代的处理更加频繁，且必须借助于写屏障，而只有当年轻代回收所带来的平均回报率更高时，这些开销才值得付出。一旦年轻代对象不再拥有较高的死亡率（即如果绝大多数对象并非在年轻时死亡），则分代的回收策略可能不再适用。

分代垃圾回收所能降低的只是“期望”的停顿时间，而非最大停顿时间。不论如何，回收器最终都会发起整堆回收，对于这一最差情况下的回收停顿时间，分代的策略无法解决，当堆空间较大时问题更甚。因此分代垃圾回收并不满足对回收停顿时间有最大值限制的硬实时（hard real-time）回收的要求。

如果回收器可以通过移动对象的方式来区分年轻和年老对象，则可以简化分代回收器的实现。物理上的分区不仅可以带来较高的局部性，而且写屏障或者回收器在对年轻代进行追踪时按照空间进行判断效率更高。当然也可以使用虚拟隔离的方式，例如在对象头部通过一个标记位进行区分，或者将该标记位置于位图中。

如何对分代垃圾回收器进行参数调整不仅困扰着回收器的开发者，也困扰着最终用户。通常分代垃圾回收器会具有众多的可调参数，且参数的种类远比简单的调整堆大小要丰富得多。与此同时，每种回收器都可能需要针对特定的程序进行精细的调整。

在实现一个具体的分代垃圾回收器时，首先需要决定的可能是要不要支持两个以上的分代。选择的结果很大程度上取决于回收器未来所服务的应用程序中对象的分布模型：如果相当一部分对象会活过年轻代回收，但是在被提升到年老代后不久便会死亡，则可以增加中间分代。但根据我们的经验，大多数系统都只提供两个分代外加一个永久对象分代（或者至少默认配置如此），究其原因除了使用多分带之外，还可以通过其他一些策略来处理对象过早提升的问题。

首先，年轻代的对象提升率取决于年轻代的整体空间大小，即如果年轻代整体空间更大，则对象会有更长的时间到达死亡。某些分代回收器允许用户指定年轻代的大小，还有一些回收器允许年轻代填满整个堆，但前提条件是为其他必需空间（包括年老代以及其他必需的复制保留区）预留足够的内存。更复杂的回收器甚至可以在动态分析的基础上改变年轻代的大小，进而达到特定的吞吐量或者停顿时间要求。

其次，可以控制对象得到提升的年龄标准，进而控制年轻代的提升率。一种提升策略是

133 将某一分代回收完成之后所有的存活对象集体提升到更老的一代。这是最简单的提升策略，因为年轻代的记忆集在次级回收完成后简单地清空即可。另一种提升策略是要求对象在得到提升之前必须经历数轮垃圾回收，但此时就需要记录对象的年龄。可以在对象头部中占用几个位来记录年龄，也可以将分代划分为多个子空间，并在每个空间中存放不同年龄段的对象，还可以将两种方案相结合。常见的实现策略包括分阶系统以及在新生代中增加存活对象半区。不论是哪种策略，同一分代中的子空间都是在相同的时间进行回收的。

最后，还可以避免对某些特殊对象进行提升。许多回收器会开辟一块永久对象空间来存放一直会存活到程序结束的对象，这些永久对象通常在编译期就可以识别出来，它们包括回收器自身的数据结构，以及包含可执行代码的对象（假定这些代码无需卸载）。

对象存活率同时也会影响写屏障的开销以及记忆集的大小。较高的提升率可能导致更多的分代间指针产生，此时写屏障的性能是否会受到影响取决于其具体实现，我们将在 11.8 节进行详细介绍。写屏障可以无条件地记录指针写操作，也可以过滤掉其中与回收器无关的部分。如果记忆集使用卡表来实现，则其所需的空间与其中所记录的指针数量无关，而基于顺序存储缓冲区（sequential store buffer）或者哈希表的记忆集则无法达到这一要求。

写屏障的调用频率也取决于是否需要对各分代对象进行独立回收。如果每个分代都需要独立回收,则所有的分代间指针都需要记录。但是,如果我们可以确保在对较老分代进行回收的同时也回收所有较年轻的分代,则写屏障只需要记录从年老代指向年轻代的指针,这通常会大大减少所需记录的指针数量。需要记录的指针数量还取决于写屏障是记录被修改的对象,还是记录其中具体被修改的域。如果使用卡表来实现记忆集,则两种记录方式均可。但如果使用记录对象而非记录域的方式,包含多个分代间指针的对象只需要占用记忆集中的一个条目,从而减少了记忆集的大小。

赋值器记录分代间指针来源的方式影响着垃圾回收的开销。尽管较低的记录精度可以减少写屏障的开销,但这却会增加回收器的工作量。使用顺序存储缓冲区记录来源域的方式可能是最精确的机制,但其中可能包含重复记录。无论是记录来源对象还是使用卡表,回收器都需要对整个对象或者卡进行扫描才能找到分代间指针。

需要指出的是,分代的策略只是进行堆分区以提升回收效率的方式之一,下一章我们将介绍其他分区策略。

9.14 抽象分代垃圾回收

最后,我们将介绍如何把 6.6 节的抽象垃圾回收框架应用在分代垃圾回收上。我们曾经介绍过, Bacon 等 [2004] 将抽象追踪过程看作一种引用计数方式,即在标记对象时增加其引用计数。算法 9.1 展示了基于两个分代以及集体提升策略的传统分代垃圾回收算法的抽象。

与第 6 章其他抽象回收算法类似,分代垃圾回收的抽象算法也通过一个多集合 I 来记录年轻代对象被延迟的引用计数。记忆集可以看作是拥有从更老的分代指向年轻分代的指针集合,而多集合 I 中所记录的条目则与记忆集中的条目一一对应,正因如此, `decNursery` 方法才需要把即将被覆盖的槽从多集合 I 中移除。如果某一年轻代对象 n 存在于多集合 I 中,则分代回收器会将其保留。对象 n 在多集合 I 中出现的次数相当于其被年老代对象所引用的次数。追踪式回收器可以使用一个位来替代对象 n 的引用计数。

[134]

当执行 `collectNursery` 方法时,多集合 I 的初始值是年轻代中引用计数非零的对象集合,当然此处的引用计数只考虑了来自年老代对象的引用。此时的多集合 I 也相当于延迟引用计数中的零引用表。当 `rootsNursery` 方法将来自根集合的引用增加到多集合 I 之后, `scanNursery` 方法从多集合 I 开始进行追踪,最后由 `sweepNursery` 方法执行清扫。清扫过程将会把存活对象从年轻代移出并提升到年老代,同时将所有不可达的年轻代对象(也就是抽象引用计数为零的对象)回收。需要注意的是,算法 9.1 的第 18 行所执行的操作是将所有年轻代存活对象集体提升到年老代,对此处的操作进行修改便可以模拟其他提升策略。

算法 9.1 抽象分代垃圾回收(回收过程)

```

1  atomic collectNursery( $I$ ):
2      rootsNursery( $I$ )
3      scanNursery( $I$ )
4      sweepNursery()
5
6  scanNursery( $W$ ):
7      while not isEmpty( $W$ )
8           $src \leftarrow$  remove( $W$ )
9           $\rho(src) \leftarrow \rho(src)+1$                 /*  $src$  从白色变为灰色 */
10         if  $\rho(src) = 1$ 

```

```

11     for each fld in Pointers(src)
12         ref ← *fld
13         if ref in Nursery
14             W ← W + [ref]
15
16 sweepNursery():
17     while not isEmpty(Nursery)
18         node ← remove(Nursery)           /* 集体提升 */
19         if  $\rho(\text{node}) = 0$                 /* 节点为白色 */
20             free(node)
21
22 rootsNursery(I)
23     for each fld ∈ Roots
24         ref ← *fld
25         if ref ≠ null and ref ∈ Nursery
26             I ← I + [ref]
27 New():
28     ref ← allocate()
29     if ref = null
30         collectNursery(I)
31         ref ← allocate()
32     if ref = null
33         collect() /* 使用追踪式回收、引用计数回收等策略进行整堆回收 */
34         ref ← allocate()
35         if ref = null
36             error "Out of memory"
37      $\rho(\text{ref}) \leftarrow 0$                     /* 新分配的对象为黑色 */
38     Nursery ← Nursery  $\cup$  {ref}          /* 在新生代中分配 */
39     return ref
40
41 incNursery(node):
42     if node in Nursery
43         I ← I + [node]
44
45 decNursery(node):
46     if node in Nursery
47         I ← I - [node]
48
49 Write(src, i, ref):
50     if src ≠ Roots and src  $\notin$  Nursery
51         incNursery(ref)
52         decNursery(src[i])
53     src[i] ← ref

```


其他分区策略

上一章我们介绍了分代垃圾回收以及其他基于对象年龄的回收策略，这些算法均按照对象的年龄将其分区，并在回收时刻依照某种年龄特征选择一个分区进行回收，例如分代垃圾回收器会优先回收最年轻的分区（或者分代）。尽管这一策略在许多应用程序中都表现得十分高效，但它并不能解决回收器所面临的所有问题。在基于年龄的回收框架之外，本章将介绍其他基于堆空间分区的回收策略。

本章将首先介绍最常见的一种分区策略，即为大对象划分出单独的空间；然后再介绍如何基于对象图的拓扑结构来进行堆划分；接下来将分析在线程栈或者特定区域进行分配的可能性；最后我们将讨论混合式堆分区算法，以及在不同时间或使用不同算法对不同空间进行回收的策略。

10.1 大对象空间

大对象空间是最常见的堆分区策略之一。可以将“大”的标准基于对象的绝对大小来定义（例如大于 1024 字节 [Ungar and Jackson, 1988]，或者相对于分配器所使用的内存块的大小 [Boehm and Weiser, 1988]），也可以相对于堆的大小 [Hosking 等, 1992]。大对象满足我们在第 8 章所介绍的多个分区标准：其分配成本通常较高，且更容易产生内存碎片（包括内部碎片以及外部碎片），因此值得为其使用一些特殊的管理策略（即使所选择的策略不太适合管理较小对象）。如果将大对象分配在以复制方式进行回收的空间，则复制保留区的空间开销可能会非常大，同时复制大对象的开销也十分高昂（如果对象本身包含较大的指针数组，则复制的开销主要取决于更新指针域及其子节点的处理开销）。正是由于这些原因，大对象空间通常使用那些不会在物理上移动对象的算法来管理。对于非移动式算法所带来的内存碎片问题，可以考虑偶尔对大对象空间进行整理 [Lang and Dupont, 1987; Hudson and Moss, 1992]。

大对象空间的实现及其管理方式存在多种解决方案。最简单的方案是采用第 7 章所介绍的空闲链表分配器外加标记-清扫回收器进行回收，另外也可以将非移动式大对象空间与包括复制式算法在内的多种回收算法相结合。有些方案将大对象拆分为（可能是固定大小的）头部和主体 [Caudill and Wirfs-Brock, 1986; Ungar and Jackson, 1988、1992; Hosking 等, 1992]，主体部分保存在非移动的区域，而头部则可以与其他小对象采用相同的方式管理。头部也可以使用分代垃圾回收器进行管理，但研究者们对于是否应当提升大对象头部仍存在分歧（如果不提升，则当大对象死亡后，回收器便可尽快回收其所占用的较大空间 [Ungar and Jackson, 1992]）。还有一些 Java 虚拟机（包括 Sun 的 ExactVM [Printezis, 2001]，Oracle 的 JRockit 以及 Microsoft 的 Marmot [Fitzgerald and Tarditi, 2000]）并不针对大对象开辟独立的空间，而是直接将其分配在年老代，这是因为大对象通常都会存活一定的时间，而预分配策略可以节省提升过程的复制开销。

10.1.1 转轮回收器

对象的复制或者移动也可以是逻辑上的而非物理上的。本节我们将介绍转轮回收器，下一节我们将介绍如何在操作系统的支持下逻辑地“移动”对象。根据三色抽象法则，追踪式回收器可以将堆中对象划分为四个集合：黑色（已完成扫描）、灰色（已访问到但尚未完成扫描）、白色（尚未访问到）以及空闲内存。回收器在追踪过程中不断对灰色集合进行处理，直到该集合变空为止。不同的回收算法对不同集合的处理方式不同。转轮回收器 [Baker, 1992a] 虽然是非移动式回收器，但它同时又兼具半区复制算法的一些优点。虽然该回收器原本是作为一种增量回收器来设计的，但它同时也可以较好地应用于万物静止式回收中的大对象管理。

转轮回收器以环状双向链表来组织对象（见图 10.1），从逆时针方向来看，环的组成依次是黑色分段、灰色分段、白色分段以及空闲分段。在整个堆中，黑色与灰色分段构成了目标空间，而白色分段相当于来源空间。回收器的操作需要用到四个指针：指针 *scan* 指向灰色分段的起始地址，同时也是灰色与黑色分段的分界点（与第 4 章的 Cheney 扫描类似），指针 *B* 和指针 *T* 分别指向白色来源空间的尾部和首部，指针 *free* 则是黑色分段与空闲分段的分界点。

在执行万物静止式回收之前，所有对象均为黑色，且均位于目标空间中。新对象的分配是通过顺时针方向移动指针 *free* 完成的，该操作相当于是从空闲分段中摘下一个内存单元并将其插入到黑色分段的首部。当指针 *free* 与指针 *B*（即来源空间的尾部）重合时，表示空闲内存耗尽，需要进行垃圾回收，此时整个转轮中最多只包含黑白两种颜色的对象。回收器首先将来源空间与目标空间互换，然后将黑色对象重新着为白色，并将指针 *T* 与指针 *B* 互换。接下来，回收器将采用类似半区复制的方式进行回收：当完成某一灰色对象的扫描后，回收器逆时针移动指针 *scan*，该操作相当于是将已完成扫描的对象插入到黑色分段的尾部。当扫描到来源空间中的白色对象时，回收器将其从白色分段中摘除并插入灰色分段。当指针 *scan* 与指针 *T* 重合时，表示灰色分段为空，即回收完成。

转轮回收器具有诸多优势。其分配与“复制”速度相当快，并发转轮回收器可以通过将对象插入到合适分段的方式简单地分配任意颜色的对象。由于摘除与插入操作并不会在物理上移动对象，因此分配与“复制”的操作可以在常数时间内完成，且与对象的大小无关。将

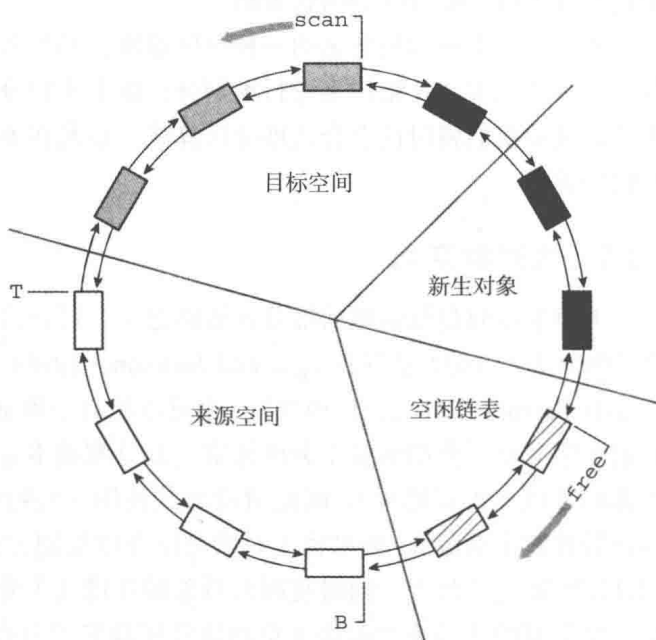


图 10.1 转轮回收器。回收器使用环状双向链表来维护对象。整个链表被划分为四个分段，每个分段都是一种颜色的对象集合。每种颜色的对象都可以从其所属的分段中摘除，并重新插入到其他分段中。转轮回收器的控制指针与其他增量复制回收器的相同 [Baker, 1978]：当指针 *scan* 与指针 *T* 重合时，表示扫描过程结束，而当指针 *free* 与指针 *B* 重合时，表示堆空间耗尽

对象插入指定分段的操作简化了我们在第 4 章中所讨论的遍历顺序选择问题：如果将对象插入到灰色分段的尾部（即在指针 τ 之前），则遍历操作就会遵从广度优先的顺序，而如果将灰色对象插入到灰色分段的首部（即指针 scan 的位置），则遍历操作会遵从深度优先的顺序，且无需额外的辅助栈。不论使用哪种遍历顺序，转轮回收器的双向链表结构都已经自然地提供了遍历所需的数据结构（即栈或者队列）。

如果将转轮回收算法当作通用的垃圾回收器来使用，则其缺点之一在于每个对象都需要引入两个指针以实现双向链表。但相对于复制式回收而言，转轮回收算法却无需任何复制保留区（因为它无需在物理上实现对象的复制），从而弥补了双向链表的空间开销。转轮回收算法的另一个问题在于如何容纳不同大小的对象（参见 [Brent, 1989; White, 1990; Baker 等, 1985]），一种解决方案是为每种空间大小使用不同的转轮 [Wilson and Johnstone, 1993]。但无论如何，这些不足之处对于大对象的管理都不会造成太大的问题。大对象转轮回收器（如 Jikes RVM 中所使用的）通常会为每个对象分配专属的页（或者一组连续的页），如果将双向链表指针保存在该页中，则它们可以复用页中的碎片空间（这些碎片是由于将对象占用的空间向上圆整到页的整数倍造成的）。另一方面，也可以将链表指针从其所属对象的页中移出并集中保存，其优势在于它不仅有助于降低用户代码破坏回收器元数据的风险，而且可以降低高速缓存不命中以及换页的开销。

10.1.2 在操作系统支持下的对象移动

如果操作系统支持，则回收器在“复制”或者“整理”对象时甚至有可能避免在物理上真正地移动它们。要达到这一目的，分配器首先必须为每个大对象分配专属的页，当需要“复制”或者“整理”某一对象时，回收器可以对其所在页进行重映射来达到更新虚拟内存地址的目的，从而避免逐字节的内存复制 [Withington, 1991]。基于操作系统的支持也可以实现大对象的增量初始化^①，即当需要清空某一大对象的内存时，不是将其一次性清零，而是修改其所在页的内存保护策略，任何尝试访问该对象未初始化部分的操作都会触发页保护陷阱，此时可以通过陷阱处理函数解除赋值器所访问地址的页保护并将该页清零，也可参见 11.1 节对清零操作的进一步讨论。

[139]

10.1.3 不包含指针的对象

对大对象进行分区管理的思想也可以用于具有其他特征的对象。如果某一对象内部不包含指针，则回收器无需对其进行扫描。基于分区信息，回收器根据对象的地址便可以简单地判定其内部是否包含指针。如果对象的标记位保存在额外的位图中，则回收器根本无需访问对象本身。如果能将较大的位图与字符串保存在独立的区域，并使用特殊的扫描器对其进行管理，即使这些区域的空间不大，程序的性能也可以得到显著提升。例如，Ungar 和 Jackson[1988] 仅凭借一个 330KB 的独立空间便将回收停顿时间降低了 3/4，而这一空间开销相对于现代标准几乎微不足道。

10.2 基于对象拓扑结构的回收器

回收器还可以按照对象图中由指针构成的拓扑结构来组织堆中对象的排列，基于这一思想可以设计出多种新的垃圾回收算法，我们将在本节对其进行讨论。

^① 参见 <http://www.memorymanagement.org/>。

10.2.1 成熟对象空间的回收

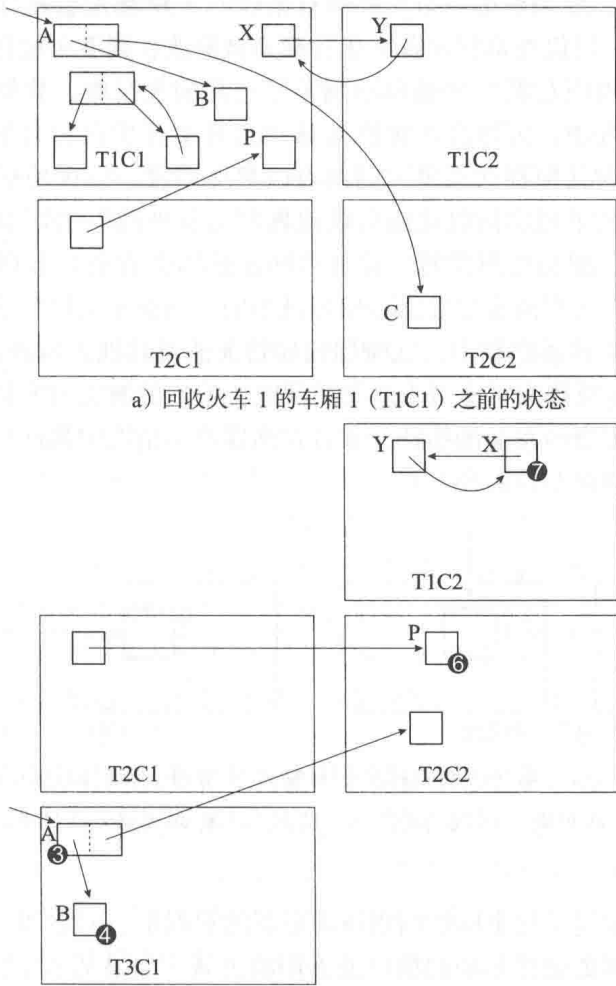
分代垃圾回收器的设计目的之一是降低回收停顿时间。回收年轻分代所需的停顿时间可以通过控制年轻代整体大小的方式进行调整,但回收最老分代所需的工作量则取决于堆中存活对象的整体大小。正如我们在第 9 章所看到的, Beltway. X. X 分代回收器 [Blackburn 等, 2002] 尝试将每次回收的工作量限制为回收带中固定大小的回收增量,但这又引入了另一个问题,即如果环状垃圾太大以至于一个回收增量无法将其完全容纳,其将无法得到回收。Bishop[1977] 和 Beltway. X. X.100 分别引入了一个大小不受限制的区域/回收增量来确保回收器的完整性,但这却违背了每次仅回收固定大小的内存空间这一设计初衷。

在基于年龄的划分策略之外, Hudson 和 Moss[1992] 另辟蹊径,提出了成熟对象空间 (mature object space, MOS) 管理方案。该方案依然将空间划分为多个固定大小的区域,每次回收只对一个区域进行处理,并将其中的存活对象复制到其他区域。Hudson 和 Moss 将每个区域称为车厢 (car), 然后使用多个先进先出链表对车厢进行组织,并称之为火车 (train), 因此该算法又俗称为火车回收器。对于每节正在处理的车厢,回收器会按照一定的规则将其中每个存活对象复制到特定的目标车厢中。该策略可以确保环状垃圾最终都会被复制到一列单独的火车中,而该列火车可以被当作垃圾进行集体回收。算法的处理过程如下:

[140]

- 1) 选择编号最小的火车 t , 并取其中编号最小的车厢 c 作为来源车厢。
- 2) 如果没有任何赋值器根引用火车 t 中的对象,且 t 的记忆集为空,则该列火车中的所有对象均不可达,回收器可以将整列火车回收,回收结束。否则继续进行步骤 3)。
- 3) 将来源车厢 c 中所有被根集合引用的对象复制到火车 t' 的目标车厢 c' 中,其中 t' 的编号比 t 大,且 t' 可能是一列新创建的火车。
- 4) 递归地将车厢 c 中从目标车厢 c' 可达的对象复制到 c' 中,如果 c' 已满,则在火车 t' 中创建一节新的车厢,并将其作为目标车厢。
- 5) 将年轻代存活对象提升到它们的引用来源所在的火车。
- 6) 扫描来源车厢 c 的记忆集,如果其中的某一对象 o 从其他火车可达,则将对象 o 复制到该列火车。
- 7) 将来源车厢 c 中的剩余可达对象复制到我所在火车 t 的最后一节车厢中,如果该车厢已满,则增加新的车厢。

算法的第 2 步中,如果一列火车只包含垃圾,即使其中仍包含跨车厢的指针结构(例如环状垃圾),回收器也会将其整体回收。由于火车的记忆集为空,所以该列火车中的任何对象都不会被其他火车引用。第 3 步和第 4 步会将来源车厢中所有直接从根集合可达或者经由本节车厢内部的指针链从根集合可达的对象移动到其他火车中,这些对象必然都是存活的,因此这两步会将它们从当前火车中其他可能是垃圾的对象中分离出来。如在图 10.2 中,回收器将位于火车 T1 车厢 C1 的对象 A 和 B 复制到新创建的火车 T3 的第一节车厢中。算法最后两步的目的在于将链式垃圾结构与其他存活对象分离。其中,第 6 步将来源车厢中从其他列车可达的对象移动到对应的火车中,如将对象 P 移动到火车 T2 的 C2 车厢中。而第 7 步则是将来源车厢中其他潜在的存活对象(如对象 X)移动到本列火车的末尾。算法的各个步骤以这种方式进行安排是十分必要的,因为某一对象可能会从多列火车可达。第 7 步完成后,车厢 c 中的剩余对象必然无法从任何外部车厢可达,因而回收器可以像半区复制策略那样将整个来源车厢回收。



a) 回收火车 1 的车厢 1 (T1C1) 之前的状态

b) 完成对 T1C1 的回收之后堆的状态。由于对象 X 被对象 Y 引用，所以回收器将其移动到对象 Y 所在的车厢。对象 A 和 B 被移动到一列新的火车 T3 中。在下一轮回收过程中，火车 T2 将被隔离并整体回收。图中带数字的标签表示对象是在算法的哪一步被复制到目标车厢中的

图 10.2 火车回收器

见 Jones[1996] 一书，已得到重印授权

火车算法存在诸多优势。该算法的回收过程是增量式的，且每个回收周期所需复制的数据量不会超过一节车厢。另外，该算法尝试将存活对象复制到你引用来源所在的车厢。由于存活对象的复制总是从编号较小的火车 / 车厢到编号较大的，所以记忆集只需要记录从编号较大的火车 / 车厢指向编号较低的火车 / 车厢的指针。如果在成熟对象空间之外引入额外的年轻分代，且在每个回收周期都对其进行回收，则记忆集也无需记录任何来自年轻分代的指针。

不幸的是，火车回收器在对赋值器一般行为的适应方面存在一些问题[⊖]，这表现在以下几个方面。将环状垃圾隔离在单独的列车中可能需要经历多个回收周期，且所需的回收周期数与环状垃圾所分布车厢总数的平方成正比。同时，火车算法在某些情况下可能会无法正常工作。以图 10.3a 所示的情况为例，假设单独一节车厢的空间不足以同时容纳 A 和 B 两个

⊖ 火车算法在 Java 5 之后的 Sun Microsystems 的 JDK 中被废弃，取而代之的是另一个停顿时间更短的并发回收器。

对象（或者指针结构），则当对第一节车厢进行回收时，对象 A 会被移动到同一列火车末尾新创建的那节车厢中。假设在本例中各个指针都未被修改，则下一轮回收将会发现首节车厢存在一个外部引用，因而对象 B 会被移动到编号更高的火车中。类似地，第三轮回收将会发现对象 A 引用了对象 B，因而会将对象 A 移动到对象 B 所在的火车中。此时，编号最低的一列火车将不再包含任何存活对象，因而可以整体回收。在这种情况下，火车回收器可以正常工作，但是，如果每次回收完成后赋值器都会将外部引用切换到第二节车厢中的对象（即图 10.3b 所示），则会出现问题：首节车厢永远不会有来自本列火车之外的引用，因而无论在哪次回收中，火车回收器都会在编号最小的一列火车末尾创建一节新的车厢，并将首节车厢中的存活对象移动到其中，此时回收器将无法对其他火车进行处理。Seligmann 和 Grarup[1995] 将这种情况称为“徒劳无功”的回收，他们的解决方案是进一步为每节车厢记录来自同一列火车中更靠后车厢的指针，并且在出现这一情况时据此将对象移动到其他火车上，从而最终实现此列火车的回收。

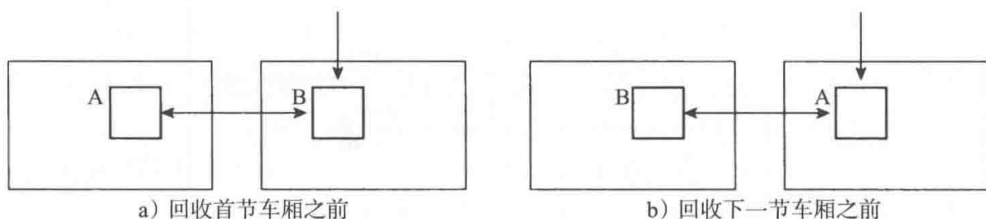


图 10.3 “徒劳无功”的回收。第一次回收将会把对象 A 移动到一节新创建的车厢中，此后赋值器却将外部引用的目标从对象 B 切换到对象 A。此时火车将回到第一次回收之前的状态，因而回收工作无法继续下去

火车回收算法仅限制了每个回收周期所需复制的数据量，但它却无法进一步限制其他一些回收相关的工作，例如记忆集的扫描以及引用的更新。如果某节车厢存在富引用对象（即该对象被引用的次数较多），则其记忆集通常较大，如果将其移动到其他车厢，回收器将不得不对大量对象的指针域进行更新。Hudson 和 Moss 建议将富引用对象移动到最新一列火车末尾专门为其创建的车厢中，后续回收过程中回收器便可对富引用车厢仅做逻辑上而非物理上的移动，进而避免大量的指针更新操作。不幸的是，这一策略却无法保证回收器能够将环状垃圾隔离到单独一列火车中。即使允许在每节富引用车厢中容纳多个富引用对象，回收器也有必要将其中的对象彼此分离，除非其中的所有对象属于同一个指针结构。Seligmann 和 Grarup[1995] 以及 Printezis 和 Garthwaite[2002] 都发现富引用对象在实践中十分普遍，后者的解决方案是允许记忆集增大到某一阈值（即 4096 个指针），并且在外部指针数量大于该值时使用哈希函数将其中的所有指针重新映射到相同大小的集合中，从而实现记忆集的扩大。Seligmann 和 Grarup 对可以回收到的垃圾进行动态评估，并尝试在此基础上降低回收频率（如果在评估时发现可以回收到的垃圾较少，则尝试降低回收频率），但 Printezis 和 Garthwaite 却发现，许多程序中通常都会存在少量包含长寿对象且长度很长的火车，这会导致 Seligmann 和 Grarup 的策略失效。

10.2.2 基于对象相关性的回收

记忆集的管理方式将显著影响火车算法在时间和空间方面的开销。对于基于分区策略的回收器而言，如果可以减少甚至彻底消除跨分区指针，回收器的性能将会得到提升。在前面

的章节中我们介绍过, Guyer 和 McKinley [2004] 使用静态分析的方法直接将新创建的对象预分配到可能与其相关的对象所在的分代, Zee 和 Rinard [2002] 在分代回收器中使用静态分析的方法去除新创建对象初始化过程中的写屏障。Hirzel 等 [2003] 对基于对象相关性的分配与回收策略做了进一步研究, 他们发现, Java 对象的生命周期与它们之间的相关性有着十分密切的联系: 仅被栈槽引用的对象寿命通常较短, 而从全局变量可达的对象则极有可能存活到程序运行结束时(他们同时指出, 这一特性同时也在很大程度上取决于对“长寿”和“短命”的精确定义)。另外, 以指针链的形式相互关联的对象通常会在同一时间死亡。

[143]

Hirzel 等 [2003] 基于这一观察结果开发出一种新的回收模型, 即基于相关性的(垃圾)回收(connectivity-based (garbage) collection, CBGC), 该模型包含四个基本组件。保守式指针分析器用于将对象图划分为稳定的分区: 如果对象 A 可能指向对象 B, 则它们将位于同一个分区, 或者由分区构成的有向无环图(directed acyclic graph, DAG)中会存在一条从 A 所在分区指向 B 所在分区的边。尽管分区的数量可能会增加(例如有新的类系加载到程序中), 但现有分区永远不会分裂。因此, 一旦回收器完成对某一分区所有来源分区的回收, 便可进一步对该分区进行回收。回收器按照拓扑顺序来选择待回收分区, 这存在两个好处: 一方面, 系统不再需要任何形式的写屏障或者记忆集; 另一方面, 在使用拓扑顺序进行遍历时, 一旦回收器完成对某一分区中对象的追踪, 则该分区内部或者其来源分区中所有的白色对象将都变成垃圾, 因此该策略具有较高的回收及时性。另外, 该算法也可以忽略对富引用子分区的处理。

Hirzel 等人发现, 对于基于对象相关性的垃圾回收器, 其性能在很大程度上取决于保守式指针分析器的分区质量、对各分区存活对象的评估结果, 以及待回收分区的选择方式。它们使用模拟程序进行实验, 其中回收器基于对象及其域的类型进行分区, 基于分区中对象从全局变量或者栈的可达性来估算分区中存活对象的量(并使用一个基于分区年龄的衰减函数来调整估算的结果), 使用贪婪算法来选择待回收分区, 但不幸的是, 实验结果令人失望, 尽管其标记/构造率在某些情况下优于半区复制回收器, 但却比 Appel 式分代回收器要差得多。另外, 该算法的最差停顿时间通常较小。与其他从分区策略中获取较高收益的回收器相比, 该回收器显然在性能上与它们有较大差距, 这一差距可能需要通过寻求更好的配置方式才能弥补。基于对象分配位置(即进行对象分配的代码地址——译者注)的动态分区策略也可能提升回收器性能, 但这又需要重新引入写屏障来实现分区的合并。

10.2.3 线程本地回收

降低垃圾回收停顿时间的一种方式是将回收器线程与赋值器线程并发执行。该策略的一个变种是增量式地执行回收工作, 即回收工作在赋值器的执行间隙穿插进行。这两种方案中, 赋值器和回收器之间需要进行大量同步操作, 因而增加了回收器的实现复杂度(我们将在后面的章节中描述增量回收器与并发回收器)。如果我们可以确保某个对象集合只会被单个赋值器线程所访问, 同时这些对象都存在于线程本地堆中, 则对这些对象的操作可以免去同步操作的开销, 从而可以将万物静止式的回收限制于单个线程之内。本节我们将对不同的线程本地回收策略进行介绍。需要注意的是, 线程本地回收方法无法处理可能共享的对象, 对它们进行处理时依然需要挂起所有的赋值器线程, 或者使用更加复杂的并发回收/增量回收技术。

线程本地回收的关键在于如何将仅可能被单个线程访问的对象与潜在的共享对象隔离。

144

线程本地回收算法通常会将堆划分为一个共享空间以及一组线程本地堆，同时算法会对指针方向有着很严格的要求：线程本地对象中的指针仅应当指向同一个线程本地堆中的对象或者共享对象，共享对象中不应当包含指向线程本地对象的指针，同时线程本地对象也不应当包含指向其他线程的本地对象的指针。可以使用静态指针分析方法实现对象的静态分区，也可以使用动态分区，但这就需要赋值器在运行时检测出违背上述指针方向要求的操作。注意，线程本地堆中对象的组织可以使用多种策略（如扁平式管理策略或者基于分代的管理策略）。另外，还可以基于对象自身来表示其是否属于共享对象（例如让对象头部中的一个标记位作为标识——译者注）。

Steensgaard [2000] 使用快速但保守的指针分析方法来判断哪些 Java 对象可能会从全局变量或多个线程可达，Ruf[2000] 也使用过类似的方法，他使用流不敏感（flow-insensitive）但上下文敏感（context-sensitive）的逃逸分析将创建对象的函数特化（specialise），并据此决定是将对象分配在线程本地堆还是共享堆。每个堆都包含一个年轻分代以及一个年老分代。Steensgaard 将所有静态域都作为线程本地堆的根，且每次回收都需要一个全局的线程汇聚（rendezvous），因而从严格意义上讲，该策略只能算是主体线程本地（mostly thread-local）回收器。回收开始时，回收器首先需要使用一个线程来完成所有从全局变量以及线程栈直接可达的对象的复制，然后再对共享堆进行 Cheney 扫描，最后再恢复各个线程，并由每个线程完成其本地堆的回收。当多个线程同时对共享堆中尚未复制的对象进行处理时可能发生冲突，在这种情况下就必须引入全局的锁。

要实现线程本地对象以及共享对象的静态分区，就需要对整个程序进行分析，这对于允许动态加载类的语言来说将会成为问题：如果某个类在静态分析完成之后才加载到程序中，则程序很可能会调用该类中未经静态分析的多态方法来创建对象，并将其赋值给某一全局可达的域，从而造成引用“泄漏”。Jones 和 King 解决了这一问题并设计出一种真正的线程本地回收器 [King, 2004 ; Jones and King, 2005]，他们基于 Steensgaard 的方法研究出了组合式逃逸分析技术，支持 Java 的动态类系加载，且可以确保在静态分析完成之后的类系加载操作依然安全。对于在 Solaris 系统中运行在多处理器上的 ExactVM Java 虚拟机，该方案会在长期执行的 Java 程序中启动一个后台线程执行分析，且速度相当快。他们为每个线程开辟了两块线程本地堆，一个用于分配确定只会被当前线程访问的对象（不管未来是否会有新的类系加载到程序中），另一个用于分配乐观本地对象（optimistically-local objects），此类对象在执行静态分析时只被一个线程访问，但在引入新的类系之后却有可能成为共享对象。纯粹的线程本地对象通常较少，它们通常不会逃逸出创建它们的方法，但乐观本地对象则相当普遍。该方案对 Steensgaard 的指针方向要求做了适当的拓展：纯线程本地对象可以引用乐观本地对象，但反之则不允许，同时乐观本地对象可以引用全局对象。图 10.4 展示了该方案中允许出现的指针方向。Jones 和 King 的方案允许对每个

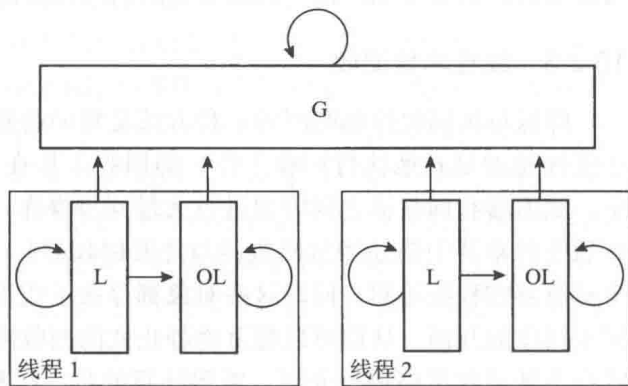


图 10.4 线程本地堆的组织方式。图中展示了纯线程本地堆 (L)、乐观本地堆 (OL) 以及共享堆 (G) 之间所有合法的指针方向 [Jones and King, 2005]

145

线程进行独立回收，从而无需引入全局的线程汇聚。如果在静态分析完成之后没有新的类系引入，则对纯线程本地堆以及乐观本地堆可以同时进行回收。一旦有新类系被动态引入，后台分析线程不但要对其方法进行特化，而且还要判断该类是否会影响现有的已经完成分析类系，同时还要确定该类的方法是否会导致原有的乐观本地分配成为多线程共享分配。如果出现这样的情况（实际应用中这些“不合格”的方法通常极少出现），则所有可能调用该方法的线程都必须将其乐观本地堆标记为共享，同时不再允许对其进行线程本地回收（它们必须和共享堆一起进行回收）。

Steensgaard 使用静态分析的方法来划分对象，但这个方法需要引入全局线程汇聚；Jones 和 King 的方案也使用静态逃逸分析，但其回收过程却是纯线程本地的，该方案同时也会对动态加载的类系进行检查，并会针对可能导致线程本地对象变成共享对象的方法进行处理。除此之外，也可以在运行时动态检测某一对象是否会逃逸出创建该对象的线程。Domani 等 [2002] 使用线程本地分配缓冲区来创建对象，但同时使用写屏障来精确捕获对象的逃逸行为。由于该方案不会将共享对象与线程本地对象分配在不同的空间，所以回收器需要一个独立的位图来记录对象的状态。当某一线程创建指向由其他线程所创建对象的引用之时，写屏障不仅要设置目标对象在位图中对应的标记位，还要将其递归闭包中的所有对象都设置为共享。在 Domani 等人所设计的并行标记-清扫回收器中，各线程可以独立进行回收，只有当系统无法完成大对象的分配，或者需要分配新的缓冲区时，才需要挂起所有线程。他们同样也会将已知的、通常会全局可达的对象（如线程对象或者类对象，或者由离线分析器判定为全局的对象）分配在单独的共享区域。回收器应当确保在线程本地回收的执行过程中不会发起全局回收，这就需要在两者之间引入适当的同步机制，我们将在后续章节中描述其中必要的握手过程。

如果所有对象都是不可修改的（immutable），则线程本地回收的实现将更加简单。Erlang [Armstrong 等，1996] 是一种严格的动态类型的函数式编程语言，基于 Erlang 的应用程序通常会使用大量轻量级进程（extremely light-weight processes）（此处的“进程”与操作系统的进程无任何关联，此处的各“进程”会共享包括地址空间在内的大量资源——译者注），且它们彼此之间通过消息传递来进行异步通信。最初的 Erlang/OTP 运行时系统是以轻量级进程为核心的（process-centric），即每个轻量级进程拥有其本地内存区域。由于 Erlang 不允许破坏性的赋值操作，因而消息传递必须使用复制语义，因此各线程可以独立地对其本地堆进行回收。这一设计方案的开销在于其消息传递操作是 $O(n)$ 时间复杂度的（其中 n 是消息的大小），且消息数据会在多个轻量级进程之间出现冗余。

为减少消息传递过程中的复制开销，Sagonas 和 Wilhelmsson 在上述架构中引入了两个共享区域，一个用于保存消息，另一个用于保存二进制数据 [Johansson 等，2002；Sagonas and Wilhelmsson，2004；Wilhelmsson，2005；Sagonas and Wilhelmsson，2006]。他们对线程本地空间以及共享消息空间之间合法的指针方向做了限制：共享消息空间中不会包含任何环状引用数据，而共享二进制数据空间则不会包含任何引用。它们使用一个静态消息分析器来引导分配过程：如果分析器推断待分配数据很可能是消息的一部分，则将其分配到共享堆中，否则便将其分配到线程本地堆中。所有的消息参数都会被封装到一个按需复制（copy-on-demand）的操作对象中，该对象会检测各消息参数是否已经存在于共享堆中，并且仅在不存在时才进行复制（编译器通常会将这一检测过程优化掉）。基于 Erlang 语言的复制式消息传递语义，分析器既可以高估对象共享情况，也可以低估。线程本地堆使用分代式的、停止-复

[146]

制式 Cheney 回收器进行管理, 并使用分代式栈扫描 [Cheng 等, 1998]。由于共享二进制对象不会形成环, 所以可以使用引用计数对其进行管理。每个轻量级进程需要维护一个簿记表^① (remembered list) 来记录其所引用的二进制对象, 这样才能确保在轻量级进程死亡时其所引用的二进制对象都能正确地减少引用计数。共享消息空间通过增量标记-清扫回收器进行管理, 其回收需要使用一定的全局同步操作。我们将在第 16 章讨论增量标记-清扫回收。

线程本地/共享区域 (thread-local/shared region) 这一内存架构最早由 Doligez 和 Leroy[1993] 提出。在他们的方案中, 本地/共享区域同时也在回收器中扮演着年轻/年老分代的角色, 该方案是针对 Concurrent Caml Light (一种支持并发原语的 ML 实现) 设计的。与 Erlang 不同, ML 语言中存在可修改对象, 因此为确保每个线程可以独立回收其年轻分代, 必须将可修改对象保存在共享的年老分代中。如果更新某一可修改对象的操作导致其引用了线程本地年轻分代中的对象, 则写屏障必须将目标对象及其递归闭包中所有的年轻代对象提升到年老代。与 Erlang 语言类似, 该方案中年轻代对象都不可修改, 因而允许其存在多个副本。在将年轻代对象复制到年老代的过程中, 回收器 (此时回收器的角色是由写屏障扮演的) 会在对象的原始副本中记录转发地址 (即其复制的共享副本的地址), 该地址会在后续的线程本地年轻代对象回收中用到。需要注意的是, 由于对象在年轻代中的副本仍在使用中, 所以在记录转发地址时不能破坏性地覆盖对象数据, 而必须使用对象头部中一个保留的域。共享堆通过并发标记-清扫回收器进行管理, 因而年老代对象可以省略这一保留域。尽管这一额外的域给年轻分代带来了一定的空间开销, 但这通常在可接受范围内, 因为年轻代对象在整个堆中所占的比例通常会比年老代对象小得多。

10.2.4 栈上分配

一些研究者建议, 任何情况下都应当尽可能在栈上而非堆中分配对象。尽管研究者们提出了多种不同的方案, 但真正实现的却很少, 用于生产系统的则更是寥寥无几。栈上分配存在多种优势: 它可以潜在降低垃圾回收的频率; 在栈上分配对象无需进行昂贵的扫描或者引用计数操作; 栈上分配在理论上应当具有较高的高速缓存友好性。但其不足之处在于, 在栈帧中分配的对象寿命很可能会被延长, 进而长期占用栈空间^②。

栈上分配技术的关键在于, 如何才能确保栈上分配的对象不会被其他寿命更长的对象引用。可以通过保守式逃逸分析来达到这一目的 (例如 [Blanchet, 1999; Gay and Steensgaard, 2000; Corry, 2006]), 也可以在运行时利用写屏障捕获逃逸对象。Baker[1992b] 最早提出 (但并非最早实现) 以一个独立进行垃圾回收的堆作为上下文 (context) 来实现栈上分配。如果栈沿着远离堆的方向增长, 则我们可以使用一个高效的、基于地址比较的写屏障来判断哪些堆中的对象会比其所引用的栈上对象存活得更久。一旦发生这样的情况, 便需将栈上对象复制 (即“懒惰分配”) 到堆中。该方案还需引入一个读屏障来处理由这一复制过程所产生的转发地址。还有一些研究者建议将栈上对象分配在独立于调用栈 (call stack) 之外的栈帧中。Cannarozzi 等 [2000] 使用写屏障来进行堆的划分, 且将每个分区与可能引用该分区的最老的活动记录相关联, 但不幸的是, 该方案 (在 Sun handle-based JDK 1.1.8 中) 的开销巨大: 每个对象都需要额外引入 4 个 32 位的字。Qian 和 Hendren [2002] 使用懒惰帧分配的策略以

① 注意不要将这一概念与分布式引用计数系统中的引用列表混淆, 后者保存的是所有引用了其所对应目标对象的进程列表。

② 只有在线程退栈时才可能将其销毁。——译者注

避免分配出空帧，并且在帧中对象发生逃逸时将该帧打上全局共享标记位。在这种情况下，写屏障同时也需要对分配该对象的代码地址做共享标记（即标记此处分配的对象可能会成为共享对象——译者注），但这就需要在对象头部记录分配该对象的代码地址。他们复用对象头部中的锁相关域来存放这一地址，但其代价是一旦对某一帧中对象加锁，则整个帧都必须被打上全局共享标记。不幸的是，库代码通常会包含许多冗余（即局部的）的锁操作（正因如此偏向锁（biased locking）才显得十分高效）。Corry[2006]使用开销更低的过程内逃逸分析技术，该方案将对象帧与循环而非函数调用相关联，从而可以较好地动态类系加载、反射、工厂方法等进行处理。

Azul 系统的多核多处理器 Java 设备可以在硬件层面支持对象逃逸检测。当在栈上分配一个对象时，指针中的某几位将被用于记录其所处的帧在栈中的深度。指针加载操作会忽略这些位，但指针写操作却会对其进行检测：如果将较新帧中对象的引用写入到较老的帧中，则会触发一个陷阱，陷阱处理函数会移动对象并修正其所有的引用来源（该对象只可能被其他更新的帧所引用）。修正操作的开销较大，因而只有此类情况很少发生时才可以确保栈上分配的效率。如果某一对象的栈上分配可能导致帧过大，则 Azul 会将该对象分配到额外的溢出区域中。Azul 发现，它们仍需要借助于偶尔执行的线程本地回收来处理长寿帧中已经死亡的栈上分配对象。

[147]

综上所述，大多数栈上分配策略到目前为止都仍未实现，即使那些宣称已经实现的算法通常也缺乏相对系统的细节，或者并未取得显著的性能提升。不可否认的是，在许多应用程序中很大一部分对象都可以使用栈上分配，且它们中的大多数通常都十分短命（Azul 发现在大型 Java 应用程序中一半以上的对象可以进行栈上分配），但这却正是分代垃圾回收可以充分发挥优势的场景。栈上分配究竟是否可以降低内存管理的开销，目前尚无定论。栈上分配的另一个问题在于，它会将整个对象都置于高速缓存中，从而减少内存带宽，即使高速缓存足够大，这一情况也不会得到优化。一种有效的解决方案是纯值替换（scalar replacement）或者对象内联（object inlining），即以局部变量来替代对象的域 [Dolby, 1997; Dolby and Chien, 1998, 2000; Gay and Steensgaard, 2000]。面向对象程序中迭代器的实现便是纯值替换的一种典型应用场景。

10.2.5 区域推断

栈上分配在更加通用的、基于区域划分的内存管理策略中属于一种受限的形式。基于区域划分来管理内存的基本出发点在于，如果将对象分配在不同区域中，则一旦某一区域中的所有对象都不再被程序使用，回收器便可立即将整个区域回收。区域的回收通常可以在常时间内完成。何时需要创建一个区域、应当将对象分配到哪个区域、何时需要对区域进行回收，这些工作可以由开发者自己实现，也可以由编译器或者运行时系统来完成，也可以将三者相结合。例如，开发者可能需要添加一些显式的指令或者注释来创建、回收区域，或者强制要求将对象分配到某一区域。最知名的显式系统可能是 RTSJ（Real-Time Specification for Java）。除了标准堆之外，RTSJ 提供了一个永久性区域以及多个受限区域，该系统同时对合法的指针方向做出限制：外层受限区域中的对象不允许引用内层受限区域中的对象。

还有一些基于分区的系统放松了对指针方向的要求，也就是说，即使某一区域中的对象仍被其他存活对象所引用，回收器也可以将其回收，但是为了确保安全，必须确保赋值器不会访问指向已回收区域的悬挂指针（dangling pointer）。此类系统通常都需要在编译期推断

出应当将对象分配到哪个区域，或者何时才能安全地回收区域，或者对开发者的注释进行检测（可能在非标准系统中）。其中最著名的当属为标准 ML 所设计的全原子化区域推断系统 [Tofte 等, 2004]。如果谨慎使用，该系统可以提升程序的性能并减少内存的使用量，但该系统也严重依赖于开发者的编程风格，同时要求开发者对区域推断算法有着很深的理解（尽管无需理解其具体实现）。在区域推断系统中，即使用户对代码进行很小的修改，也会引发推断结果的显著变化，这在无形中增加了开发者对算法的理解难度以及程序的维护难度。ML Kit 的推断算法在大型程序中开销巨大（例如，编译一个仅 58 000 行的程序就需要花费一个半小时）。Tofte 等人建议，最好的实践方案是仅将区域推断用于已经深入理解的编程模式上，而其他部分的管理则应当交由垃圾回收器。

148

10.3 混合标记 - 清扫、复制式回收器

在对内存块中的存活对象进行处理时，Spoonhower 等 [2005] 引入两个阈值来判断是应当将其复制，还是对其进行标记 - 清扫。当内存块中存活对象的比例小于迁移阈值 (evacuation threshold) 时，则将其复制到其他内存块，而当内存块中的空闲内存大于分配阈值 (allocation threshold) 时，该内存块才可以用于分配。这两个阈值决定着何时以及如何减少内存碎片。例如，标记 - 清扫回收器的迁移阈值为零（即任何情况下都不会复制存活对象），但分配阈值为 100%（即内存块中的任何空闲内存都可以用于分配），而半区复制回收器的迁移阈值为 100%，且其分配阈值为零（在下次回收之前，来源空间将不会用于内存分配），如图 10.5 所示。过于消极（即迁移阈值和分配阈值都较低）的内存管理器会受到内存碎片问题的影响，而过于积极（即迁移阈值和分配阈值都较高）的管理器则会存在较大的性能开销，因为它需要对数据进行复制，或者需要更多的堆遍历过程。

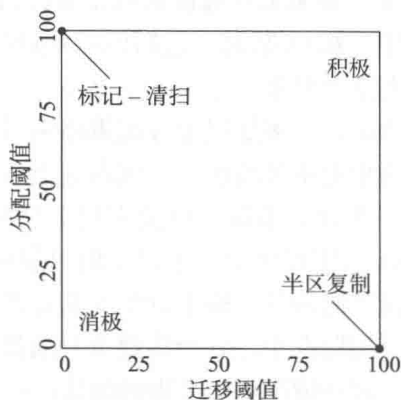


图 10.5 追踪式回收器的连续性。Spoonhower 等人对比了追踪式回收器的迁移阈值和分配阈值，前者表示内存块中存活对象达到多小的比例时才使用复制的方式进行回收，后者表示内存块中的空闲内存达到多大比例时才可以用于内存分配

Spoonhower et al [2005], doi: 10.1145/1064979.1064989.

© 2005 Association for Computing Machinery, Inc., 经许可后转载

大型或者长期运行的应用程序很容易受到内存碎片问题的影响，除非使用整理式回收器来管理堆内存。但与非移动式回收器相比，整理操作无论是在时间上还是空间上都存在较大开销。半区复制算法需要一个额外的复制保留区，而标记 - 整理算法则需要进行多次堆遍历才能完成对象的移动。为解决这一问题，Lang 和 Dupont [1987] 提出将标记 - 清扫回收与半区复制相结合，同时在堆中进行增量内存整理的方案（即一次只整理一个内存区域）。该方案将整个堆空间划分为 $k + 1$ 个大小相等的窗口，其中包含一个空窗口。回收开始时，回收器选定某个窗口作为来源空间，并以空窗口作为目标空间，而其他窗口则使用标记 - 清扫策略进行管理。在追踪过程中，回收器会将来源窗口中的存活对象复制到目标窗口中，同时对其他窗口中的对象进行标记（见图 10.6）。与此同时，回收器必须确保将所有窗口中指向来

149

源窗口的引用更新到目标窗口中的对应副本里。

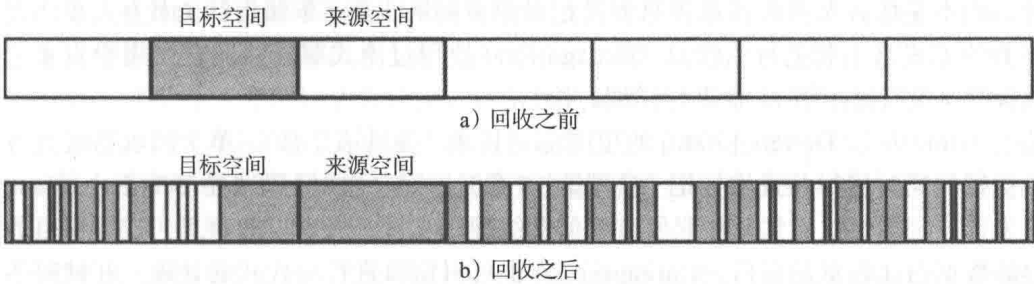


图 10.6 渐进式增量整理垃圾回收。回收器将某一窗口（来源空间）中的存活对象复制到灰色方框所示的空窗口（目标空间），其他窗口则在原地回收。每次回收完成后，来源 / 目标窗口的索引号将向前递增，最终便可实现整个堆的整理

Jones [1996], 经许可后转载

通过这种一次复制一个窗口的方式，Lang 和 Dupont 可以通过 k 次回收实现整个堆的整理，其空间开销仅为可用堆空间的 $1/k$ 。与标记 - 整理算法不同，该方案无需额外的堆遍历过程或者其他额外数据结构。他们同时发现，即使目标空间使用 Cheney 算法进行管理，整体算法的追踪顺序也具有一定的柔性：在追踪阶段的每一步，回收器都可以从标记 - 清扫和半区复制两个工作列表中选择一个来获取对象，不过 Lang 和 Dupont 建议优先处理标记 - 清扫回收的工作列表，这样不仅有助于限制标记栈的大小，而且标记 - 清扫回收通常会比 Cheney 扫描具有更好的局部性。

Spoonhower 等 [2005] 为 C# 所设计的回收器则使用更加富有弹性的方案。该方案使用内存块容量预测技术来判定某一内存块是需要进行原地回收，还是需要使用复制式回收，这可以使用静态预测方法（如大对象空间所占用的内存块）并使用固定的迁移阈值（分代回收器假定存活下来的年轻对象数量很少），也可以使用动态预测或者动态阈值（即在追踪阶段进行判定）。Spoonhower 等人根据上次回收的结果来预测适合某一内存块的回收方式（被钉住对象的内存块只能进行原地回收），进而无需在回收过程中引入额外的遍历过程。Dimpsey 等 [2000]（稍后将对其进行描述）也使用类似的策略，但它们需要维护一个空闲间隙链表，并在其中进行阶跃指针分配。

10.3.1 Garbage-First 回收

Garbage-First[Detlefs 等, 2004] 是一种精密且复杂的增量整理算法，其目的在于满足软实时性能要求，即在任意 y 毫秒的时间切片中，花费在垃圾回收上的时间均不超过 x 毫秒。该算法在 Sun 微系统 JDK 7 HotSpot VM 中引入，并将以长期演进的方式逐渐替代原有的并发标记 - 清扫回收器，从而达到更加可预测的整理响应时间要求。本节我们仅关注该算法的分区策略。

与 Lang 和 Dupont 的回收器类似，Garbage-First 也将堆空间划分为数个虚拟地址连续的、大小相等的窗口。该算法在整体上存在一个用于内存分配的、来自空窗口列表的当前窗口。为减少多个赋值器线程之间的同步，每个线程都拥有本地顺序分配缓冲区，而这一缓冲区本身是使用 CompareAndSwap 原子操作从当前分配窗口中分配出来的。大对象也可简单地从前分配窗口中直接分配，特别大的对象（即大于单个窗口 $3/4$ 的对象）则会从其专属的窗口序列中进行分配。

与 Lang 和 Dupont 的方案不同，Garbage-First 允许选择任意窗口进行回收，这便要求赋

值器写屏障记录所有跨窗口指针的创建。特别需要注意的是，此处需要记录的是所有跨窗口的指针，而不是像火车回收器那样只需要记录单方向跨火车 / 车厢指针（因为火车回收器会以可预测的方式对车厢进行回收）。Garbage-First 使用过滤式写屏障，它使用卡表来记录回收相关指针（我们将在第 11 章进行详细讨论）。

基于 Printezis 和 Detlefs [2000] 的位图标记技术（参见第 2 章），单个回收器线程可以在赋值器执行的同时进行并发堆标记（参见第 16 章）。一旦完成标记过程，Garbage-First 便通过位图来选择定罪窗口，然后挂起所有赋值器线程以实现定罪窗口的整理。定罪窗口通常是那些存活数据占比较低的窗口。Garbage-First 也可对窗口进行分代式的处理。在纯粹的“全年轻”模式下，定罪窗口将是那些在上一次回收之后用于分配的窗口。而在“部分年轻”模式下，回收器可以额外地将一些窗口增加到定罪窗口集合。无论在哪种分代模式下，写屏障都可以过滤掉来自年轻代的指针。与其他策略类似，Garbage-First 试图识别出富引用对象并将其隔离在专属的窗口中，此类窗口永远不会被加入到定罪窗口集合中，因而也无需任何形式的记忆集。

10.3.2 Immix 回收以及其他回收

接下来我们将介绍 3 种回收器，它们都通过付出一定的时间或空间开销来解决标记 - 清扫回收的碎片化问题。每种回收器都通过一种不同的方式来解决如下 3 个问题：如何最好地利用堆空间、如何避免对去碎片化操作（复制或标记 - 整理）的依赖、如何降低回收器循环的时间开销。

Dimpsey 等 [2000] 为 IBM 服务器的 Java 虚拟机 1.1.7 版本设计了一种复杂的并行标记 - 清扫（偶尔执行整理操作）回收器。与 Sun 的 1.1.5 版本回收器类似，该方案也使用线程本地分配缓冲区^①，小对象将直接在该缓冲区中进行顺序分配，缓冲区本身以及大对象（比缓冲区大小的 1/4 还大的对象）则使用空闲链表分配并需要一定的同步操作。Dimpsey 等人发现，如果仅依赖这一架构，回收器的性能会非常差。大多数空闲链表的分配需求都是为线程申请新的本地分配缓冲区，但靠近链表头部的空闲内存单元通常无法满足这一分配需求，从而导致较长的查找时间。为解决这一问题，他们额外引入了两个空闲链表，一个仅用于分配线程本地缓冲区（1.5KB 外加缓冲区头部），另一个则用于分配超过缓冲区大小且小于 512KB 的对象。一旦用于分配线程本地缓冲区的空闲链表为空，则分配器从另一个大对象链表中分配一个大块内存，并将其分割成多个缓冲区。这一优化大大提高了 Java 应用程序在单处理器上的执行性能，对于多处理而言效果更佳。

Dimpsey 等人使用额外的位图来标记对象，在清扫阶段对位图进行遍历时，回收器可以一次检测一个字节或者一个字。他们同时对清扫过程进行了一定的优化：他们引入两个表以快速计算位图中任意一个字节所对应的前导与尾部为零位，清扫器会避免对较小的连续垃圾空间进行处理，并通过对象头部中的一个标记位来区分大对象与较小的连续垃圾空间。在回收完成后，分配器只会从新的缓冲区中进行顺序分配，即使某一分配缓冲区中存在部分可用空间，分配器也不会从其中分配任何对象。这一策略不仅可以减少清扫时间，同时也缩短了空闲链表的长度，因为其中不再包含任何小块空闲内存。

[151]

这一策略的潜在开销在于，回收器并未将某些空闲内存归还给分配器。但由于对象通常“成簇创建，成批死亡”，因而 Dimpsey 等人可以尽量少地依赖整理过程。根据 Johnstone

① 与我们使用的术语不同，Dimpsey 将其称为“线程本地堆”。

[1997] 的建议, 他们使用基于地址顺序的首次适应分配策略以增加在可用堆中创建足够大的空洞的几率。另外他们还允许在线程本地分配中使用可变大小的内存块: 如果空闲链表中第一个用于线程本地分配的缓冲区小于某一期望值 T (即 6KB), 则线程将直接使用该缓冲区 (注意该缓冲区不应小于空闲链表允许插入的最小空间大小); 如果其大小介于 T 和 $2T$ 之间, 则将其拆分为两个大小相等的缓冲区; 否则将从该缓冲区中分裂出一个大小为 T 的缓冲区。Dimpsey 等人还在堆中预留 5% 的空间以用于拓展块保护 [Korn and Vo, 1985], 即仅在回收完成之后可用空间依然不足的情况下才动用这些内存。

与 IBM 服务器中的方案实现类似, Immix 回收器 [Blackburn and McKinley, 2008] 也通过这一方式避免内存碎片化。该回收器也属于主体标记-清扫回收器, 但其在必要情况下消除碎片的方法是复制式而非整理式回收。Immix 回收器与本节所介绍的其他回收器一样使用块结构堆。回收器将堆空间划分为 32KB 的内存块, 这不仅是线程为本地分配缓冲区申请空间的单元, 也是执行碎片整理的操作单元。每次回收过程中, 回收器会根据上次回收的结果来预测哪些内存块需要进行原地回收, 哪些内存块需要进行复制式回收 (与 Spoonhower 等人的方法类似, 但与 Detlefs 等人的方法不同, 后者基于并发标记来进行预测)。IBM 服务器中的回收器以及 Immix 回收器都使用速度较快的顺序分配策略, 不同之处在于前者减少碎片的方式是从大小可变的缓冲区中进行分配, 而后者允许在部分可用的缓冲区中以行 (line) 为单位的间隙里进行分配, 行的大小通常为 128 字节, 大致与高速缓存行的大小匹配。Dimpsey 等人对清扫过程的优化方法是忽略对较小连续垃圾空间的处理, 而 Blackburn 和 McKinley 则是以行为单位来回收可复用内存块中的空间。下面我们将介绍 Immix 回收器的实现细节。

Immix 回收器同时支持从完全为空以及部分为空 (即可复用) 的内存块中进行分配。图 10.7 展示了可复用内存块的结构。Immix 回收器将对象划分为大对象 (它们将从大对象空间中分配)、中等对象 (即大小超过一个行的对象) 以及小对象, 大多数 Java 对象都是小对象。算法 10.1 展示了 Immix 回收器分配小对象或者中等对象的方法。Immix 回收器优先将对象分配到可复用内存块的空隙中, 其所使用的分配算法为线性循环首次适应分配法。在算法的快速路径中, 分配器尝试在当前连续空闲行序列中进行顺序分配 (第 2 行), 如果失败, 则区分小对象和中等对象并执行不同的分配策略。

[152]

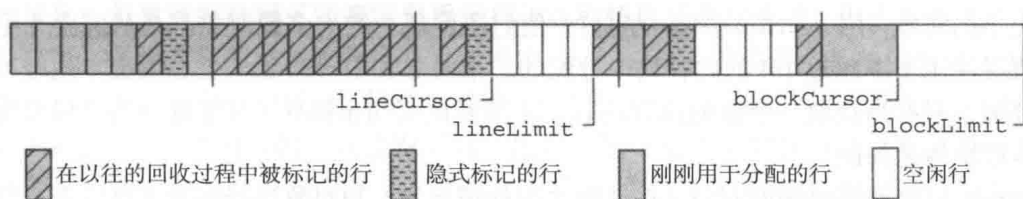


图 10.7 Immix 回收器中的内存分配。图中展示了一个内存块 (block) 中的行 (line)。Immix 回收器使用顺序分配策略从可复用内存块中分配小对象, 即 `lineCursor` 向着 `lineLimit` 的方向移动, 两者汇合之后再将它们分别移动到下一个空闲行序列的首尾。中等大小的对象则使用顺序分配策略在完全为空的内存块中分配。Immix 回收器需要同时对行和对象进行标记。由于小对象可能会跨越两个行 (但不可能跨越更多行), 所以 (显式) 标记行之后的一行都会得到隐式标记, 分配器不会从这些行中进行分配

Blackburn and McKinley [2008], doi: 10.1145/1375581.1375586.

© 2008 Association for Computing Machinery, Inc., 经许可后转载

算法 10.1 Immix 回收器中的内存分配

```

1  alloc(size):
2      addr ← sequentialAllocate(lines)
3      if addr ≠ null
4          return addr
5      if size ≤ LINE_SIZE
6          return allocSlowHot(size)
7      else
8          return overflowAlloc(size)
9
10 allocSlowHot(size):
11     lines ← getNextLineInBlock()
12     if lines = null
13         lines ← getNextRecyclableBlock()
14         if lines = null
15             lines ← getFreeBlock()
16             if lines = null
17                 return null /* 内存耗尽 */
18     return alloc(size)
19
20 overflowAlloc(size):
21     addr ← sequentialAllocate(block)
22     if addr ≠ null
23         return addr
24     block ← getFreeBlock()
25     if block = null
26         return null /* 内存耗尽 */
27     return sequentialAllocate(block)

```

我们先考虑小对象的分配策略。分配器先在当前内存块中查找下一个空闲行序列（第 11 行），如果失败，则尝试从下一个可复用内存块（第 13 行）的空闲行或者下一个空内存块（第 15 行）中进行分配。如果后续两个尝试均失败，则发起垃圾回收。需要注意的是，与首次适应分配不同，分配器永远不会在部分填充的行中分配对象。

在大多数应用程序中，少量 Java 对象的大小可能会超过一行，但不会太大。Blackburn 和 McKinley 发现，如果对这些对象使用与小对象相同的方式进行处理，则会浪费大量行。因此，为避免在可复用内存块中引入碎片，他们使用顺序分配策略直接从空闲内存块中分配中等大小的对象（用 `overflowAlloc` 方法）。他们还发现，绝大多数对象都是从完全为空的内存块或者使用率不超过 1/4 的内存块中分配的。小对象和中等对象均是在线程本地缓冲区内进行分配，只有当获取一个新的内存块时，才需要使用同步操作（对于部分为空以及完全为空的内存块均是如此）。

Immix 回收器需要同时对行（作者称之为标记区域）和对象进行标记（对后者进行标记是为了确保扫描过程的正常结束）。从定义上来看，小对象所占用的空间必然小于一行，但它仍有可能跨越两个行，因此 Immix 回收器会对小对象占据的第二个行进行隐式（且保守）标记，即所有位于某个已标记行之后的行都会被分配器忽略（见图 10.7）。这样便造成在最差情况下，每个间隙内部都可能会有一行被浪费。Blackburn 和 McKinley 发现，如果在扫描对象时（而不是标记对象同时将其添加到工作列表时）标记其所在的行进行有助于提升追踪过程的性能，因为开销更大的扫描操作可以掩盖对行进行标记的延迟。与小对象不同，回收器会对中等对象所在的行进行精确标记（根据对象头部的一个位来区分小对象和中等对象）。

Immix 回收器只需偶尔执行整理操作，且整理过程可以在标记过程中进行。是否需要

整理取决于某个描述堆碎片化程度的统计变量, 清扫器在每次回收完成后都会设置该变量。Immix 回收器根据每个内存块中空隙和已标记行的数量来衡量碎片化程度, 并在下一次回收过程中选择碎片程度最高的内存块作为备选整理对象。由于统计变量只是作为整理过程的一个参考, 所以回收器可以在空间不足的情况下中止整理过程。在实际应用中, Immix 回收器在大多数基准测试程序中都不太需要进行整理。

10.3.3 受限内存空间中的复制式回收

上述各种增量回收技术都仅需要一个内存块作为复制保留区, 且需要经历多次回收才能完成整个堆的整理。Sachindran 和 Moss[2003] 将这一技术应用在内存空间受限环境下的分代回收器中。他们所设计的 Mark-Copy 回收器将年老代划分为一组连续的内存块, 但在进行整堆回收时, 该回收器可以一次完成多个内存块中存活对象的迁移, 而非一次只处理一个内存块。与其他分代回收器类似, 用于对象分配的新生区回收频率较高, 其中的存活对象会被提升到年老代。如果堆中只剩一个空闲内存块, 则会发起整堆回收。

如果回收器可独立回收每个内存块, 则必须记录所有内存块之间的指针, 这将使写屏障的设计更加复杂, 因为原本只需记录跨代指针而现在还需要记录跨内存块的指针。因此 Mark-Copy 回收器会在标记阶段为每个内存块构造单向记忆集, 并计算其中存活对象总量。将记忆集的构造任务从赋值器转移到标记阶段有两个好处: 第一, 记忆集本身可以十分精确(因为记忆集可以仅包含在回收时刻从编号较高内存块指向编号较低内存块的指针), 且不包含任何重复记录, 因此回收器可以沿着内存块编号从小到大的方向(目的是避免在记忆集中记录双向指针), 将一组连续内存块中的存活对象迁移到空闲块中; 第二, 因为标记阶段已经完成了对每个内存块中存活对象总量的计算, 所以回收器可以准确评估出当前回收过程能完成多少个内存块中存活对象的迁移, 例如图 10.8 中的第二次遍历过程可以完成连续 3 个内存块中存活对象的迁移。回收完成后, 回收器会将已完成迁移的内存块释放(解除其内存映射)。

与标准的半区复制回收器相比, Mark-Copy 回收器显著增加了可用空间的比例, 同时在空间大小相同的情况下其回收频率也会降低。该回收器也可设计成增量式的, 即将年老代内存块的回收穿插在年轻代回收之间。该回收器同时也存在一些缺点: 每次整堆回收都需要两次扫描年老代对象, 一次用于标记, 另一次用于复制; 回收器需要预留额外的空间来实现标记栈以及记忆集; 每次复制过程可能都需要重新扫描线程栈以及全局变量。但不可否认的是, 与其他需要更多保留空间的复制式分代回收器相比, Mark-Copy 回收器在某些场景下表现更佳。

[154]

Mark-Copy 回收器要求各内存块在地址空间上连续, 而 MC² 回收器 [Sachindran 等, 2004] 则通过将内存块编号的方式放宽了这一限制, 这带来几个好处: 已完成复制的内存块不必再通过解除内存映射的方式释放, 进而避免了在 32 位环境下耗尽虚拟地址空间的风险; 该方案允许通过改变内存块编号的方式实现逻辑上的复制, 这对于存活对象比例较大的内存块将十分有用(此时逻辑复制所带来的收益会大于复制或者整理); 对内存块进行逻辑编号的方式也允许回收器在回收过程中改变内存块的回收顺序。与 Mark-Copy 回收器不同, MC² 回收器将复制年老代内存块所需的遍历过程分摊在年轻代回收中, 它同时也使用 Steele 插入式屏障对年老代进行增量标记(我们将在第 15 章讨论增量标记)。借助于增量标记技术, 回收器可以在内存耗尽之前的某一时刻开始对年老代的回收, 并且可以适应性地调整每个增量的工作量, 进而避免内存耗尽时可能出现的较大停顿。与本章所介绍的其他回收器类似, MC² 回收器将富引用对象隔离在特殊的内存块中, 同时省去对这些内存块记忆集的维护(因

此富引用对象会被当作永久性对象来对待，但回收器仍可将其恢复为一般对象)。另外，为了限制记忆集的大小，回收器还会将较大记忆集的实现方式从顺序存储缓冲区转化为卡表(我们将在第 11 章介绍这些技术)，大数组也通过卡表的方式进行管理，其实现方式是将大数组的卡表紧邻其末尾保存。通过对多种技术的精细化整合，MC² 回收器最终成为一种空间利用率高、吞吐量大、停顿时间较为平衡的回收器。

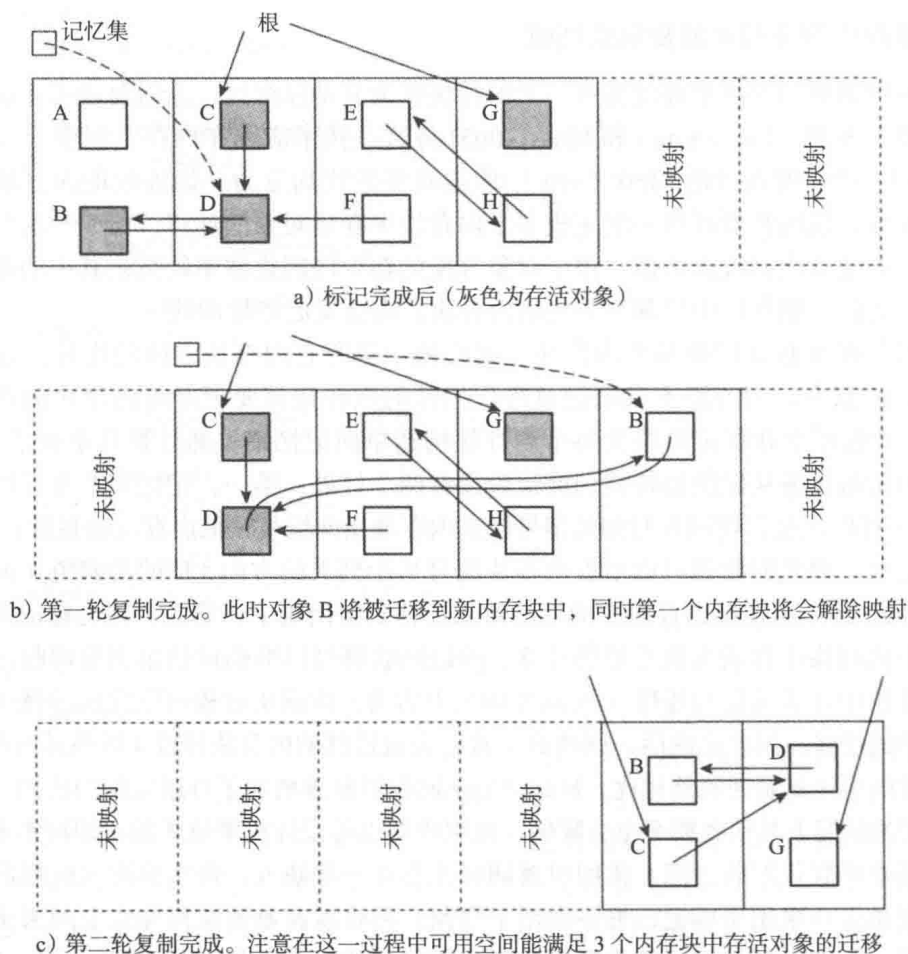


图 10.8 Mark-Copy 回收器将年老代划分为多个内存块。回收器将在标记阶段完成记忆集的构建，其中记录的是跨内存块的指针。回收器一次可以完成多个内存块中存活对象的复制，并且在复制完成后将其释放 (解除其内存映射)

Sachindran and Moss[2003], doi: 10.1145/949305.949335.

©2003 Association for Computing Machinery, Inc., 经允许后可转载

10.4 书签回收器

上一节所述的各种增量整理技术均可以 (最终) 完成堆的整理，其回收过程不仅在时间开销方面低于传统的标记-整理回收，而且空间开销也远低于标准的半区复制回收。但是，如果堆空间过大以致于赋值器操作或者回收器追踪会产生换页行为，则程序的性能依然可能受到严重影响。换页的开销可能会超过上百万个时钟周期，因而避免程序执行过程中的缺页异常也是十分重要的。书签 (bookmarking) 垃圾回收器 [Hertz 等, 2005] 不仅可以缓解赋值

器执行过程中的缺页异常，而且能够避免回收过程中的缺页异常。

书签垃圾回收器可以通过操作系统的虚拟内存管理器来引导页淘汰策略。如果没有回收器的引导，虚拟内存管理器通常在页淘汰方面没有太多的选择余地，例如，对于半区复制回收器与使用最近最少使用淘汰策略的内存管理器同时工作的情形，在回收时间之外，被淘汰的页通常是当前未使用但很快便会被目标空间所用的页，因此如果大部分对象都十分短命，则很可能最近最少使用的页就是分配器将要使用的下一个页，而这正是最糟的一种换页情形。来源空间通常不会受换页问题的影响，不仅是因为赋值器在下一次回收过程之前不会访问该空间，而且是其中的数据也无需写回到外存中。

书签垃圾回收器可以在不引发缺页异常的前提下完成垃圾回收的追踪过程。在追踪过程中，回收器保守地假定非驻留（non-resident）页中所有对象都是存活的，但同时依然需要定位出所有从该页可达的对象。为了达到这一目的，回收器会在某一存活页被换出时对其进行扫描，找出其中所有的对外引用并为其目标对象设置书签，同时如果该页重新装载到内存，则将其对应的书签删除。回收器可以使用书签来实现追踪过程的持续。

书签回收器需要对虚拟内存管理器进行修改，即在页淘汰发生时向应用程序发送一个信号。如果分配器无法获取新的空闲页，则唤起垃圾回收器并重新从刚刚清空的页中进行分配。回收器可以通过某些特定的系统调用来影响虚拟内存管理器的行为，例如 `madvise` 系统调用以及 `MADV_DONTNEED` 位。书签回收器会在回收完成后尝试收缩堆空间以避免缺页异常，而年轻分代或者回收器元数据所在的页则绝不会被换出。如果无法找到一个空页（并将其换出），则回收器会寻找一个“牺牲品”（通常是事先预定的页）并扫描其对外引用，然后在其目标对象的头域中设置一个特殊的标记位。Herts 等人在 Linux 内核中引入了一个系统调用，以允许用户进程显式地将一组页换出。

如果整个堆都未装入物理内存，则在整堆回收开始时回收器会先扫描存在书签的对象，并将其加入到回收器的工作列表中。尽管这一操作开销较大，但在堆空间较小的情况下该策略通常会比产生一次缺页异常的开销要低。回收器偶尔需要对年老代进行整理，此时回收器会在标记阶段统计每个空间大小分级中存活对象的数量，并据此计算完成整理所需的最小的页集合。回收器使用 Cheney 扫描来将存活对象移动到选定的页上（除非对象已经在目标页上）。为了避免对非驻留页中的引用进行更新，回收器不会移动有书签的对象，进而规避由此产生的缺页异常。

155
156

10.5 超引用计数回收器

目前为止我们已经介绍了多种基于分区的堆组织方式，每种分区方式都允许在堆的不同空间中使用不同的算法或者策略，各空间也可以同时或者各自进行回收。我们可以根据对象的期望寿命或者大小进行分区，进而提升堆的使用率。在本章的最后，我们将介绍如何根据对象的修改频率来进行分区。

大量证据表明，在许多应用程序中，年轻对象的创建和死亡率都相当高，赋值器对这些对象的修改也十分频繁（例如将其初始化）[Stefanović, 1999]。对于此类对象而言，复制式回收是一种十分高效的回收策略，因为它允许顺序分配且仅需要对存活对象进行复制，而对象的存活率却往往很低。现代应用程序的堆空间以及存活数据会越来越大，且长寿对象通常具有较低的死亡率和修改频率，这些因素都会给追踪式回收器带来一定的挑战：追踪的开销正比于存活对象的总量，而频繁对长寿对象进行追踪通常会影响程序性能。与追踪式回收

器相比，引用计数策略能更好地适应这一场景，因为其开销通常仅与被修改对象的总量成正比。Blackburn 和 McKinley [2003] 认为，在为年轻代和年老代选择各自的回收策略时，应当将分代大小、分代中对象的期望寿命以及对象的变更率这三者结合起来考虑。

因此，他们所设计的超引用计数（ulterior reference counting）回收器使用复制式回收来管理年轻代，同时使用引用计数来管理年老代。年轻代的空间大小有限，且使用顺序分配策略。回收器会将所有在年轻代回收中存活的对象复制到由分区适应空闲链表管理的成熟空间。赋值器写屏障的任务有二，一是正确管理成熟空间中对象的引用计数，二是记录从成熟空间指向年轻代对象的指针。赋值器会将涉及栈槽或者寄存器的引用计数操作延迟，同时回收器会将堆中对象的引用计数操作合并。一旦写屏障发现某一未被记录的对象发生变更，则会将其添加到日志中。日志中所记录的是对象的地址，并且会为其所有位于成熟空间的子节点缓冲一次引用计数减少操作^①。

在回收过程中，垃圾回收器会将年轻代存活对象移动到由引用计数管理的成熟空间中，并且回收两个空间中的所有不可达对象，其具体过程如下。回收器首先将日志中每个子节点的引用计数加一，并且将其所有位于年轻代的目标对象标记为存活，然后将其添加到年轻代回收器的工作列表中。当完成所有年轻代对象的提升后，回收器会增加这些对象所有子节点的引用计数。与其他延迟引用计数算法类似，回收过程中直接从根可达的对象的引用计数也会临时性地加一，且所有已缓冲的引用计数增加操作都会先于已缓冲的引用计数减少操作执行。该回收器使用 Recycler 算法 [Bacon and Rajan, 2001] 来处理环状引用，但它并不会在每次回收中都对所有存在引用计数减少操作的对象执行试验删除，这一过程仅在可用堆空间小于某一用户自定义阈值时才会触发。

图 10.9 展示了超引用计数的抽象示意，我们可以将其与第 5 章图 5.1 所示的标准延迟引用计数操作进行比较。

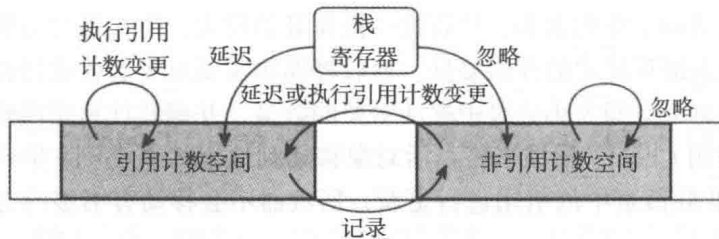


图 10.9 超引用计数原理图。该算法将堆划分为两个空间，仅有一个空间由引用计数算法进行管理。从图中我们看出，哪些情况下的指针加载或写入操作需要立即执行引用计数，哪些需要延迟，哪些可以忽略

Blackburn and McKinley [2003], doi: 10.1145/949305.949336.

©2003 Association for Computing Machinery, Inc., 经许可后转载

10.6 需要考虑的问题

通过本章的介绍我们可以知道，除了对象年龄之外，还有许多其他因素可以作为堆分区的依据。在将堆中对象分区之后，我们可以使用不同的策略或者机制来管理不同的分区或空间，且可以针对每个空间中对象的特征选择最合适的策略或机制。基于物理隔离的分区策略

^① 第 5 章中，Levanoni 和 Petrank [2001] 所设计的写屏障需要记录对象在被修改之前的快照，这与此处所描述的方案有所不同。

有诸多优点,包括基于地址的快速空间关系判定、更高的局部性、可选择的碎片整理方式,以及更低的内存管理开销。

最普遍的一种分区策略是将大对象与小对象进行区分管理,即把大对象分配在其专属的、不会移动的空间,回收器也会避免对其中的对象进行复制或者整理。大对象通常会被分配在其专属的页序列中,且这些页通常不会与其他对象共享。将不包含指针的对象(例如用于表示图像的位图)与大数组对象进行区分也是十分必要的,因为回收器根本无需对前者进行追踪,如果使用额外的位图进行标记,则回收器根本无需访问真正的对象,进而避免了潜在的缺页异常以及高速缓存不命中问题。

基于分区的策略同样还可以实现堆的增量回收,即回收器可以仅选择堆中某些子空间进行回收,就像分代回收器可以只处理年轻代那样。两者所达到的效果是相同的:回收器在一次回收过程中可以仅完成少量的、有限的工作,进而减少对赋值器的影响。

实现堆分区的一种指导思想是按照对象的拓扑结构进行分区,也就是按照赋值器访问对象的模式来进行分区。该策略的目的之一是确保较大指针结构最终会落入同一个分区中,进而可以将其整体回收,如果做不到这一点,则仅对单个分区进行回收将无法释放跨分区环状垃圾。诸如火车回收器 [Hudson and Moss, 1992] 以及基于对象相关性的回收器 [Hirzel et 等, 2003] 都属于此类回收器。火车回收器一次仅处理一个较小的空间,并且将其中的存活对象复制到了其引用来源所在的空间。基于对象关联性的回收器使用指针分析技术将对象分配到由一组分区组成的有向无环图中,并可以基于对象拓扑顺序来回收各分区。系统也可以将对象置于能够在常数时间内完成回收的区域中,一旦回收器发现其中所有对象均不可达,便可将它们整体回收,该策略可以基于显式分配的方式实现(如 Java 的实时规范),也可以使用某种区域推断算法来进行自动引导 [Tofte 等, 2004]。

158

系统还可以借助于指针分析算法将仅会被一个线程访问的对象分配到特定的空间中 [Steensgaard, 2000; Jones and King, 2005], 此时回收器便可在不停止其他线程的前提下独立回收某一线程所对应的空间。Blackburn 和 McKinley [2003] 发现,赋值器对年轻对象的修改通常会比年老对象更加频繁,因而他们所设计的超引用计数回收器使用复制的策略来管理年轻对象,同时使用引用计数来管理年老对象。复制式回收策略非常适用于管理对象死亡率很高的空间,同时由于它不会给赋值器的修改操作带来任何额外开销,所以也适用于对象变更率较高的场合。引用计数算法则适用于较大的、较稳定的空间的管理,对这些空间进行追踪通常会付出较高的代价。

另一种常见的分区策略是动态选择合适的算法来处理不同的分区 [Lang and Dupont, 1987; Detlefs 等, 2004; Blackburn and McKinley, 2008], 其目的在于实现堆空间内存碎片的增量整理,即将整理的开销分摊在多个回收周期中。每次回收过程会选择一个或者数个区域执行碎片整理,其中的存活对象通常会被复制到另一个空间中,而其他空间中的对象则会进行原地标记。与标准的半区复制回收算法相比,分区之间的复制可以减少复制保留区所需的空間。Mark-Copy 回收器 [Sachindran and Moss, 2003] 将这一思想发挥到极致,该回收器针对内存空间有限的环境而设计,其一次回收过程可以完成所有年老代对象的复制,但其复制过程依然以内存块为单元,从而将复制保留区的空间开销限制为一个内存块。MC² 回收器 [Sachindran 等, 2004] 由 Mark-Copy 回收器发展而来,并在增量垃圾回收方面取得长足进步,该回收器实现了更高的内存利用率以及 CPU 利用率,同时可以避免较长的或者较为集中的停顿时间。

159

运行时接口

自动内存管理系统的核心是回收器、分配器及其算法与数据结构，但是如果应用程序无法使用它们，或者用它们无法合理地访问底层平台，则一切都只是空中楼阁而已。某些算法对编程语言的实现具有一定的要求，例如要求其提供一定的信息，或者要求其满足特定的不变式。本章我们所关注的正式回收器（和分配器）与整个系统中其他模块之间的接口，包括编程语言及其编译器、更低级别的操作系统和代码库。

我们将依次介绍：新对象的分配，对象、全局变量以及栈中指针的查找与识别，访问或更新对象指针时所需执行的动作（即屏障），赋值器与回收器之间的同步，地址空间管理，虚拟内存的使用。

11.1 对象分配接口

从编程语言的角度来看，新分配的对象不应当仅是一块新分配的内存，它还应当初始化到编程语言及其实现所需要的程度。不同语言在对象分配方面的需求差异较大。最简单的一种情况是 C 语言，它仅要求分配器返回一块新的可用内存，内存单元中很可能是一些随机数据，因此其初始化任务就完全成为开发者的责任。诸如 Haskell 之类的纯函数式语言则属于最复杂的一种情况，它要求开发者必须在语言层面为新对象中的每个域提供初始值，因而应用程序不可能访问到任何未经初始化的对象。对类型安全关注度较高的编程语言通常要求对新分配对象的所有域进行合理初始化，域的初值可以由开发者提供，也可以使用每种类型默认的安全值，还可以将两者结合。

我们将对象的分配以及初始化过程划分为 3 个阶段，但并非所有的语言或者所有情况下都需要完成所有阶段。

阶段 1：分配一块大小合适的、符合字节对齐要求的内存单元，这一工作是由内存管理器的分配子系统完成的。

阶段 2：系统级初始化（system initialisation），即在对象被用户程序访问之前，其所有的域都必须初始化到适当的值。例如在面向对象语言中，设定新分配对象的方法分派向量（method dispatch vector）即是该阶段的任务之一。该阶段通常也需要在对象头部设置编程语言或内存管理器所需的头域，对 Java 对象而言，则包括哈希值以及同步相关信息，而 Java 数组则需要明确记录其长度。

阶段 3：次级初始化（secondary initialisation），即在对象已经“脱离”分配子空间，并且可以潜在被程序的其他部分、线程访问时，进一步设置（或更新）其某些域。

下面是 3 种不同的语言中对象分配及其初始化过程：

- C：所有分配工作均在阶段 1 完成，编程语言无需提供任何形式的系统级初始化或次级初始化，所有这些任务均由开发者完成（或者初始化失败）。需要注意的是，分配器仍需对已分配内存单元的头域进行修改，以确保能够在未来将其释放，但这一头域存在于返回给调用者的内存单元之外。
- Java：阶段 1 和阶段 2 共同完成新对象的方法分派向量、哈希值、同步信息的初始

化,同时将所有其他域设置为某一默认值(通常全为零)。数组的长度域也在这两个阶段完成初始化。字节码 `new` 所返回的对象便处于这一状态,此时尽管对象满足类型安全要求,但其依然是完全“空白”的对象。阶段 3 在 Java 语言中对应的表现形式是对象构造函数或者静态初始化程序中的代码,或者在对象创建完成后将某些域设置为非零值的代码段。`final` 域的初始化也是在阶段 3 中完成的,因此一旦过早地将新创建的对象暴露给其他线程,同时又要避免其他线程感知到对象域的变化,实现起来将十分复杂。

- **Haskell**: 开发者所提供的构造函数会完成新创建对象所有域的填充,同时编译器和内存管理器可以共同保证对象在真正可达之前能够完成初始化,因此所有初始化工作均在阶段 1 和阶段 2 中完成,不存在阶段 3 了。ML 也采用相同的方式来实现对象的初始化,但它需要把可变对象当作特例来处理。Lisp 也使用偏函数式的方法来实现创建对象,它同时也支持对象的修改。

如果编程语言要求完整的对象初始化语义(如 Haskell 和 ML),则对象分配接口的定义会存在一些细小的问题:为确保开发者能够为每种对象的每个域提供初始值,分配接口可能会存在无限种,具体取决于对象所包含域的数量及其类型。Modula-3 允许开发者提供函数式的初始化方法(并非一定要求如此),可以将初始化闭包(initialising closure)传递给分配子过程,后者会分配适当的空间并执行初始化闭包来填充对象的域,从而解决了这一问题。初始化闭包中包含了需要设置的初始值以及将其设置到对象特定域的代码。Modula-3 使用静态作用域(static scope),且闭包本身并不需要从堆中进行分配,其本身只是一个静态链指针(指向包含它的环境(enclosing environment)中的变量),因此它可以避免分配过程中的无限循环递归[⊖]。但是,如果编译器可以自动生成初始化代码,则无论初始化过程是在分配过程内部还是外部便都无关紧要了。

Glasgow Haskell 编译器采用另一种不同的策略来解决这一问题:它将阶段 1 和阶段 2 中的所有操作内联,并且在内存耗尽时唤起回收器。在创建新对象时,分配器使用顺序分配来获取内存,其实现简单,且初始化过程通常只需要使用已经计算好的值来填充对象的头部以及其他域。这是编译器与特定分配算法(以及回收算法)紧密关联的一个案例。

函数式初始化过程具有两个显著的优点:第一,它不仅可以确保完成对象的初始化,而且其初始化代码对于回收器而言属于原子操作;第二,初始化过程中的写操作可以避免某些写屏障的引入,特别是在分代式回收器中,正在进行初始化的对象必然会比其所引用的其他对象都要年轻,因而初始化过程可以忽略分代间写屏障。但值得注意的是,这一结论在 Java 的构造函数中通常不成立 [Zee and Rinard, 2002]。

[162]

语言级别的对象分配需求最终都会调用内存分配子过程,某些编译器会将这一过程内联,并完成阶段 1 的全部操作以及阶段 2 的部分或全部操作。分配过程需要满足的一个关键要求是:阶段 1 和阶段 2 中的所有操作对于其他线程以及回收器都应当是原子化的,只有这样才能确保系统的其他模块不会访问到未经系统初始化的对象。如果我们对分配器接口(阶段 1)进行深入思考便会发现,3 个阶段之间的工作划分会存在多种可能的组合方式。在分配过程中需要考虑的参数如下。

- **待分配空间大小**,通常以字节为单位,也可能以字或者其他粒度为单位。当需要分配数组时,分配接口可以将元素大小以及元素个数作为独立的输入参数。

⊖ 即分配过程需要创建一个闭包,而创建闭包又需要调用分配过程。——译者注

- **字节对齐要求**，分配器通常会以一种默认的方式进行字节对齐，但调用者也可以要求更严格的字节对齐方式。这些要求可能包括必须以 2 的整数次幂对齐（如按照字、双字、四字等进行对齐），或者在此基础上增加一个偏移量（例如在四字对齐的基础上偏移一个字）。
- **待分配对象的类别（kind）**，例如，诸如 Java 等托管运行时语言通常会将数组与非数组对象进行区分，某些系统会将不包含指针的对象与其他对象进行区分 [Boehm and Weiser, 1988]，还有一些系统会将包含可执行代码的对象与不包含可执行代码的对象区分对待。简而言之，任何需要分配器特殊对待的需求都要在分配接口中得到体现。
- **待分配对象的具体类型（type）**，即编程语言所关心的类型。与“类别”不同，分配器通常不需要关注对象的“类型”，但却会通过“类型”来初始化对象。将这一信息传递给分配子过程不仅可以简化阶段 2 的原子化实现（即将这一任务转移到阶段 1），而且可以避免在每个分配位置上引入额外的指令，进而减少代码体积。

分配接口究竟需要支持上述各种参数中的哪些，这在一定程度上取决于其所服务的编程语言。我们还可以在分配接口中传递一些冗余参数以避免运行时的额外计算。分配接口的一种实现策略是提供一个全功能型分配函数，该接口支持众多的参数并且可以对所有情况进行处理，而为了加速分配以及精简参数，我们也可以为不同类别的对象定制不同的分配接口。以 Java 为例，定制化的分配接口可以分为以下几种：纯对象（非数组）的分配、byte/boolean 数组（元素为 1 字节）的分配、short/char 数组（元素为 2 字节）的分配、int/float 数组（元素为 4 字节）的分配、指针数组以及 long/double 数组（元素为 8 字节）的分配。除此之外还需考虑系统内部对象的分配接口，例如表示类型的对象、方法分派表、方法代码等，具体的分配方式取决于是否要将其置于可回收堆中，即使它们不从托管堆中分配，系统仍需为其提供特殊接口，以从显式释放的分配器中进行分配。

阶段 1 完成后，分配器可以通过如下几个后置条件（post-condition）来检测该阶段的执行是否成功。

- 已分配内存单元满足预定的大小以及字节对齐要求，但此时该内存单元还不能被赋值器访问。
- 已分配内存单元已完成清零，这可以确保程序不会将内存单元中原有的指针或者非指针数据误认为是有效的引用。零是非常好的一个值，对于指针而言，零值表示空指针，而对于大多数类型而言，零值都是平常的、合法的值。某些语言（如 Java）需要通过清零或者其他类似的方式来确保安全类型的安全性。在调试系统中，将未分配的内存设置成特殊的非零值十分有用，例如 0xdeadbeef 或者 0xcafebabe，其字面意思就是表示其当前所处的状态。
- 内存单元已被赋予调用者所要求的类型。当然这一过程只有当调用者将类型信息传给分配器时才需考虑。与最小后置条件（即该条款中的第一条）相比，此处的区别在于分配器会填充对象的头部。
- 确保对象的完全类型安全性。这不仅涉及清零行为，而且还涉及填充对象头部的行为。这一步完成后，对象并未达到完整初始化的标准，因为此时对象中的每个域均只是安全的、平常的、默认的非零值，而应用程序通常要求将至少一个域初始化到非默认的值。
- 确保对象完全初始化。这通常要求调用者在分配接口中传递所有的初值，因而这一要求并不普遍。一个较好的例子是 Lisp 语言中的 cons 函数，该函数的调用相当普遍，因而有理由为其提供单独的分配函数，以加速并简化其分配接口。

究竟最合适的后置条件是哪一个？某些后置条件（如清零）取决于编程语言的相关语义，同样还有一些后置条件取决于其所处环境的并发程度，以及对象可能会以何种方式从其诞生的线程中“逃逸”（从而成为其他线程或者回收器可达的对象）。一般来说，并发程度越高、逃逸情况越普遍，后置条件的要求就越高。

下面我们来考虑分配器无法立即满足分配要求时应当如何处理。在大多数系统中，我们希望在分配子过程内部调用垃圾回收，并向调用者隐藏这一事实。此时调用者几乎不需要做任何事情，同时也可以避免在每个分配位置上引入重试[⊖]。然而，我们也可以将大多数情况下的快速路径（即分配成功的情况）进行内联，同时将回收-重试这一函数放在内联代码之外。如果我们将阶段 1 的代码内联，则阶段 1 和阶段 2 将不存在明显的分界线，但整个代码序列必须高效且原子化地实现。后续我将介绍赋值器和回收器之间的握手机制，其中便包括这一原子化要求的具体实现。在实现分配过程的原子化之后，我们便可将分配过程看作是仅有赋值器参与的行为。

11.1.1 分配过程的加速

许多系统和应用程序都存在较高的内存分配率，因此尽可能提升分配速度便显得十分重要。此处的关键技术之一是将一般情况下的代码（即“快速路径”）内联，同时将较少执行的、处理更复杂情况的“慢速路径”作为函数调用，而具体如何进行选择就需要在合适的负载下精心地进行比较测量。

顺序分配显而易见的优点便是实现简单，其一般情况下的代码序列较短。如果处理器的寄存器数量足够多，则系统甚至可以专门使用一个寄存器来保存阶跃指针（bump pointer），同时再使用一个寄存器来保存堆地址上限，此时典型的代码序列可能会是：将阶跃指针复制给结果寄存器、给阶跃指针增加待分配空间的大小、判断阶跃指针是否超出堆地址上限、在结果为真时调用慢速路径。需要注意的是，只有当使用线程本地顺序分配时，才能将阶跃指针保存在寄存器中。某些 ML 和 Haskell 进一步将一段代码序列中的多个分配请求合并成一个较大的请求，使得只需要进行一次地址上限判断与分支。类似的技术也可以用于其他单入口多出口的代码序列，即一次性分配所有可能执行路径下最大的内存需求，或者仅在开始执行代码序列时使用该值来做基本的地址上限判断。

164

尽管顺序分配几乎必然会比空闲链表分配要快，但如果借助于部分内联以及优化，分区适应分配也可以十分高效。如果我们可以静态计算出对应的空间大小分级，并且使用寄存器来保存空闲链表数组的地址，此时的分配过程将是：加载对应空闲链表的头指针，判断其是否为零，如果为零则调用慢速路径，加载下一个指针，将链表头指针设置为下一个指针。在多线程系统中，最后一步操作可能需要原子化，即使用 CompareAndSwap 操作并在失败时进行重试。另外也可以为每个线程提供专属的空闲链表序列，并独立对其进行回收。

11.1.2 清零

为确保安全，某些系统要求将其空闲内存设置为指定的值，该值通常是零，也可能是其他一些特殊的值（一般是为了调试）。仅提供最基本分配函数的系统（例如 C）通常都不会如此，或者仅在调试状态下才会执行这一操作。分配保障较强的系统（例如具有完全初始化能

⊖ 原则上讲，Java 应用程序可以捕获这一异常，然后将某些指针值空赋并再次尝试内存分配，但在实际应用中，我们尚未见过使用这一策略的程序。另外，Java 的软引用可能是解决这一问题的更好的方法。

力的函数式语言) 通常无需对空闲内存清零。尽管如此, 将空闲内存设置为特定的值仍会有助于系统调试。Java 便是需要将空闲内存清零的一个典型案例。

系统应当在何时执行清零操作? 如何进行清零? 我们可以在每次分配对象时将其清零, 但经验告诉我们, 一次性对较大空间进行清零将更加高效。使用显式的内存写操作进行清零可能会引发大量的高速缓存不命中, 同时在某些硬件架构上执行大量清零操作也可能影响读操作, 因为读操作必须阻塞到硬件写缓冲区中的清零操作全部执行完毕为止。某些 ML 的实现以及 Sun 的 HotSpot Java 虚拟机 (在顺序分配中) 对位于阶跃指针之前的数据进行精确地预取, 并以此掩盖新分配数据从内存加载到高速缓存时的延迟 [Appel, 1994; Gongalves and Appel, 1995], 但现代处理器通常可以探测到这一访问模式并实现硬件预取。Diwan 等人 [1994] 发现, 使用支持以字为单位 (per-word basis) 进行分配的写分配高速缓存 (write-allocate cache) 可以获得最佳性能, 但在实践中这一结论并非永远成立。

从分配器的实现角度来看, 将整个内存块清零的最佳方式通常是调用运行时库提供的清零函数, 例如 `bzero`。这些函数通常会针对特定系统进行高度优化, 甚至可能使用特殊的指令直接清零高速缓存而不将其写入内存, 例如 PowerPC 上的 `dcbz` 指令 (Data Cache Block Zero)。开发者直接使用这些指令可能较难, 因为高速缓存行的大小是与处理器架构密切相关的一个参数。任何情况下, 系统在对以 2 的整数次幂对齐的大内存块清零时通常会达到最佳性能。

另一种清零技术是使用虚拟内存的请求二进制零页 (demand-zero page)。该技术通常更适合程序启动时的场景, 如果在运行时使用该技术, 则开发者需要手工将待清零页重新映射 (remap), 操作系统会对该页设置陷阱, 并在应用程序访问该页时将其清零。由于相关操作的开销相对较大, 因而其性能可能还比不上开发者自行调用库函数清零。只有当需要清零的页面数量较多且地址连续时, 该技术的执行开销才可能得到有效掩盖, 其性能优势才能得到凸显。

与清零相关的另一个问题是何时执行清零。我们可以在垃圾回收完成之后立即进行清零, 但其显而易见的缺点便是延长了回收停顿时间, 同时还可能使大量内存被修改, 而这些内存很可能在很久之后才会用到。被清零的数据很可能需要从高速缓存写回到内存, 并在分配阶段重新加载到高速缓存。我们可能会根据直观经验武断地认为, 对内存的最佳清零时机应当是在其将要被分配出去之前的某一时刻, 这样处理器便可在分配器访问这块内存之前将其预取到高速缓存中, 但问题在于, 即使被清零的内存距离阶跃指针不远, 其依然很容易被刷新到内存中。对于现代硬件处理器而言, 很难说 Appel 所描述的预取技术能有多少效果, 或者其至少需要通过很精细的调整才能确定合适的预取范围。如果是在调试环境下, 将空闲内存清零或者向其中写入特殊值的操作应当在节点释放后立即执行, 这样我们便可以在尽可能大的时间范围内捕获错误。

165

11.2 指针查找

回收器需要通过指针查找来确定对象的可达性。某些回收算法需要精确掌握程序中所有指针的信息。特别是对于移动式回收器而言, 如果需要将某一对象从地址 x 移动到新地址 x' , 则必须将所有指向 x 的指针更新到 x' 。安全回收某一对象的前提条件是程序不会再次访问该对象, 但反之则不成立: 将程序不再使用的对象保留并不存在安全问题, 尽管这可能降低空间利用率 (不可否认, 如果程序无法获取可用堆内存, 则可能崩溃)。因此, 回收器可以保守地认为所有引用均指向了不可移动的对象, 但不应武断地移动其不能确定是否可以移动的对象。基本的引用计数算法便是保守式的。使用保守式回收的另一个原因在于回收器缺乏精确的指针信息, 因此它可能会将某一非指针的值当作指针, 特别是当该值看起来像是引

用了某一对象时。在本节接下来的内容中，我们将先讨论保守式指针查找的相关技术，然后再介绍不同位置（如对象内部、栈、寄存器等）的精确指针查找技术。

11.2.1 保守式指针查找

保守式指针查找的技术基础是将每个与指针大小相同的已对齐字节序列当作可能是指针的值，即模糊指针（ambiguous pointer）。回收器可以掌握组成堆的内存区域集合，甚至知道这些区域中哪些部分已经分配出去，因而它可以快速排除掉必然不是指针的值。为确保回收过程的性能，鉴别指针的工作必须十分高效。这一过程通常包含两个阶段。第一阶段，回收器首先过滤掉未指向任何堆空间地址的值。如果堆空间本身就是一大块连续内存，则这一过程可以通过简单的地址判断来实现，另外也可以根据模糊指针的高位地址计算出其所对应的内存块编号，并通过一个堆内存块索引表进行查找。在第二阶段，回收器需要鉴别出模糊指针所指向的地址是否真正被分配出去，这一过程可以借助一个记录有已分配内存颗粒的位图来完成。例如，Boehm-Demers-Weiser 保守式回收器 [Boehm and Weiser, 1988] 使用块结构堆，且其每个内存块仅用于分配一种大小的内存单元。内存单元的大小保存在内存块所关联的元数据中，而其状态（已分配或空闲）则反映在位图中。对于一个模糊指针，回收器首先使用堆边界来对其进行判定，然后再判断其所引用的内存块是否已被分配，如果判断成立，则进一步检测其所指向的内存单元是否已被分配。只有最后一步的判断结果为真，回收器才可以对模糊指针的目标对象进行标记。图 11.1 展示了对模糊指针进行处理的全部过程，其每次判断大约需要 30 个 RISC 指令。

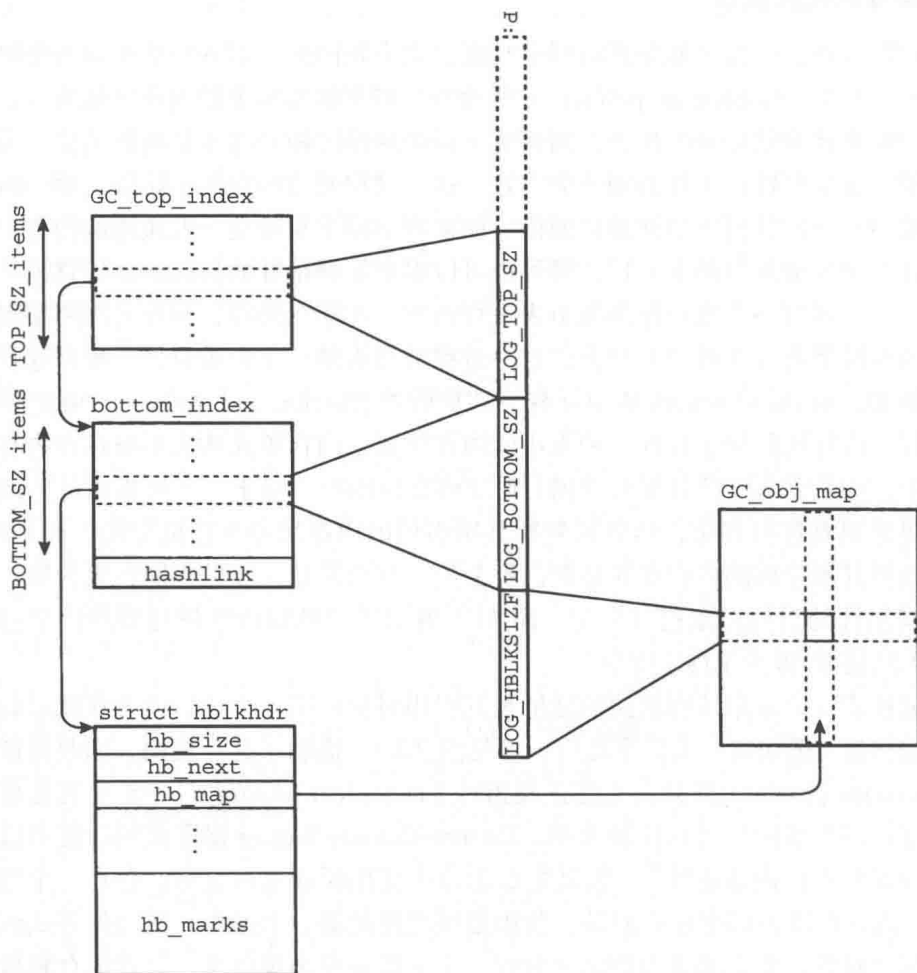
某些编程语言要求指针所指向的地址是其引用对象的第一个字，或者在此基础上增加一些标准的偏移量（例如数个头部字之后，参见图 7.2）。借助于这一规则，回收器便可忽略内部指针（interior pointer）而只需关注正规指针（canonical pointer）。不论是否需要支持内部指针，保守式回收器的设计均比较简单，Boehm-Demers-Weiser 保守式回收器可以通过配置来选择是否需要支持内部指针[⊖]。如果在 C 语言中使用保守式回收器，存在一个细节问题需要关注：C 语言允许内部指针指向某一数组范围之外的第一个元素，此时保守式回收器要么必须维护两个对象，要么必须为数组多分配一个字以避免出现歧义。显式内存释放系统可以在对象之间插入额外的头部来解决这一问题。编译器的优化可能会“破坏”指针，从而引发回收器的误判，我们将在 11.2.8 节详细讨论这一问题。

某些非指针的值可能导致回收器错误地保留一个实际上并不可达的对象，因而 Boehm[1993] 设计了黑名单（black-listing）机制来避免在堆中使用被这些非指针值所“指向”的虚拟地址空间。特别地，如果回收器断定某个模糊指针指向了未分配的内存块，可以将该内存块加入到黑名单，但必须确保永远不在其中进行分配，否则后续的追踪过程便可能将伪指针误认为真正的指针。回收器同时还支持在特定内存块中仅分配不包含指针的对象（如位图），这一区分策略不仅可以提升回收效率（因为无需扫描对象的内容），同时也可以避免昂贵的黑名单查询开销（即天然避免了将位图中的数据当作指针）。回收器还可以进一步区分非法指针是否可能是内部指针，并据此改进黑名单（如果不允许使用内部指针，则堆空间中不可能存在内部指针）。当允许使用内部指针时，黑名单中所记录的内存块在任何情况下都不得使用；而当不允许使用内部指针时，黑名单所记录的内存块可以分配不包含指针的小对

166

⊖ 该回收器在这两种模式下均支持内部指针，但在较为严格的一种场景下，回收器要求所有可达对象都必须包含一个来自于非内部指针的引用，此时标记过程会忽略内部指针所产生的引用。

象（这通常不会造成太多浪费）。在赋值器首次执行堆分配之前，回收器先发起一次回收以初始化黑名单。分配器通常也会避免使用地址末尾包含太多零的内存块，因为栈中的非指针数据通常可能会“引用”这些地址。



判断值 p 是否指向某个已分配对象：

- ① 判断 p 是否指向堆空间的最小和最大边界之间的某一地址。
- ② 获取 p 的高位地址，将其作为索引，并在第一级表中查找对应的第二级表。在 64 位地址空间下，第一级表通过链式哈希表的方式组织，而不是使用数组。
- ③ 获取 p 的中间几位，将其作为索引，并在第二级表中查找对应的内存块。
- ④ 判断“可能存在”的对象在内存块中的偏移是否为 hb_size 的整数倍。
- ⑤ 在当前内存块所对应的位图中进行查找，判断 p 所指向的内存单元是否已被分配。

图 11.1 保守式指针查找（展示了 Boehm-Demers-Weiser 保守式回收器所使用的两级查找树、内存块头部、已分配内存块集合）

见 Jones [1996] 一书，经许可后转载

11.2.2 使用带标签值进行精确指针查找

某些系统（特别是基于动态类型的系统）支持为每个值附带一个特殊的标签（tag），以表示其类型。标签的基本实现策略有二：位窃取（bit stealing）和页簇（big bags of pages）。位

窃取的方法需要在每个值中预留出一个或者多个位（通常是字的最高或最低几位），同时要求可能包含指针的对象必须以面向字（word-oriented）的方式进行布局。例如，对于一台依照字节进行寻址且每个字包含四个字节的机器，如果我们要求每个对象都必须依照字来进行对齐，则指针的最低两位必然都是零，因而我们可以将这两位用作标签。我们亦可使用其他值来表示整数，例如可以要求所有用于表示整数的值最低位都必须是 1，同时以高 31 位来表示整数的具体值（尽管这一方案确实减少了我们可以直接表达的整数范围）。为确保堆的可解析性（参见第 7.6 节），我们可以要求堆中对象的第一个字必须以二进制 10 作为低两位。表 11.1 介绍了一种标签编码方案，它与 Smalltalk 中真正使用的编码方案类似。

表 11.1 一种指针标签编码方案

标签值	编码类型
00	指针
01	对象头部
x1	整数

可能会有读者对带标签整数的处理效率提出挑战，但对于现代流水线处理器而言，这几乎不会成为问题，一次高速缓存不命中所造成的延迟便可轻易掩盖掉这一开销。为支持使用带标签整数的动态类型语言，SPARC 架构提供了专门的指令来对带标签整数直接进行加减操作，且这些指令均可以判断操作是否发生溢出。某些版本甚至还可以针对操作溢出或者被操作数低两位不为零的情况设置陷阱。基于 SPARC 架构我们可以使用表 11.2 所示的标签编码方案。该方案要求我们对指针所代表的引用进行调整[⊖]，

表 11.2 SPARC 架构所使用的标签编码方案

标签值	编码类型
00	整数
01	指针
10	其他原始值
11	对象头部

167
168

在大多数情况下，这一调整操作可以通过在加载和存储指令中引入一个偏移量来实现，但对数组的访问是一个例外：在访问数组中的某个元素时，我们需要根据数组索引号以及这一额外的偏移量来计算其最终的访问地址。真实硬件架构对带标签整数的支持进一步说明了位窃取方案的合理性：Motorola MC68000 处理器曾经使用过这一编码方案，该处理器包含一条加载指令，该指令可以通过一个基址寄存器、一个其他寄存器外加一个立即数来构造有效地址，因此在 MC68000 处理器上使用该编码方案不存在太大的额外开销。

页簇方案是将标签 / 类型信息与对象所在的内存块相关联，因此其关联关系通常是动态的且需要额外的查表操作。该方案的不足之处在于标签 / 类型信息的获取需要额外的内存加载操作，但其优势在于整数以及其他原生类型可以完全使用其原本所占据的空间。该方案意味着系统中存在一组内存块专门用于保存整数，同时还有一组专门的内存块用于保存浮点数等。由于这些纯值不可能发生变化[⊖]，因而在分配新对象时可能需要进行哈希查找以避免创建已经存在的对象。这一所谓哈希构造（hash consing）技术的历史相当悠久 [Ershov, 1958; Goto, 1974]（来自于 Lisp 语言中用于创建新对组（pair）的 cons 函数）。在使用哈希构造技术的 Lisp 实现中，分配器通过一张哈希表来维护所有不可变对组，并且会在所需对组已经存在的情况避免重复分配。该方案可以轻易推广到其他在堆中分配并管理不可变对象的场景，如 Java 语言中 Integer 类的实例。另外，哈希表和不可变对象之间使用弱引用（参见 12.2 节）进行关联可能会是比较好的一种实现方案。

11.2.3 对象中的精确指针查找

如果不使用带标签值，那么要找出对象中所包含的指针，必然需要知道对象的类型（至

⊖ 因为 SPARC 架构要求指针的低两位为 01。——译者注
⊖ 所谓的“不可能发生变化”只是这一具体方案的特性，而非具体语言的特性。使用页簇方案的开发者也可以通过另一种方式来表示整数（或者浮点数等），即使用带标签指针指向真实的（不带标签的）值。

169

少需要知道对象中的哪些域是指针域)。对于面向对象语言(确切地讲,是使用动态方法分派(dynamic method dispatch)机制的语言),指向对象的指针并不能完全反映运行时对象的类型,因而我们需要将对象的类型信息与对象本身关联,其实现方式通常是在对象头部增加一个指向其类型信息的指针域。面向对象语言通常会为每一种类型生成方法分派向量,并在对象头部增加一个指向其方法分派向量的指针,因此编程语言便可将对象的类型信息保存在方法分派向量中,或者从方法分派向量可达的其他位置。如此一来,回收器或者运行时系统中其他依赖对象类型信息的模块(如 Java 的反射机制)便可快速获取对象的类型信息。回收器需要的是一个能够反映对象内部指针域位置的表,该表的实现方式有二:一是使用与标记位图相似的位向量(bit vector),二是使用一个向量来记录指针域在对象中的偏移量。Huang 等人 [2004] 通过调整偏移向量中元素的顺序来获取不同的追踪顺序,复制式回收器可以据此按照不同的顺序排列存活对象,进而提升高速缓存性能。这一调整操作需在运行时谨慎执行(在万物静止式回收器中)。

将包含指针的对象与不包含指针的对象进行分区(partition),在某些方面来说是比较简单的一种指针识别方法。该策略在某些语言和系统的设计[⊖]中可以直接使用,但在其他语言中则可能遇到问题。例如在 ML 中,对象可以具有多态性(polymorphic)。假设某一对象在某些情况下将某个域当作指针,在另一些情况下又将该域当作非指针值,那么如果系统生成一段适用于所有多态情况的代码,则根本无法对两种情况进行区分。对于允许派生类复用基类代码的面向对象系统,子类的域将位于基类所有域之后,这将必然导致指针域与非指针域的混合。这一问题的一种解决方案是将两种不同的域沿着不同的方向排列,即指针域沿着偏移量为负的方向排列,而非指针域沿着偏移量为正的方向排列,该方案也称为双向对象布局(bidirectional object layout)。在以字节方式寻址的机器上,对于按照字来对齐的对象,我们可以将对象头部第一个字的最低位设置为 1,而按照字进行对齐又可以确保指针域的最低两位必然全为零(参见美国专利 5,900,001),从而保证了堆的可解析性。在实际应用中,扁平排列方式通常不会成为问题,而且正如 Huang 等人 [2004] 所描述的,这一方式实际上具有诸多优势。

某些系统会针对每种类型生成面向对象风格的代码,从而实现对象的追踪、复制等 [Thomas, 1993; Thomas and Jones, 1994; Thomas, 1995a, b]。我们可以将查表方式看作是类型解释器,而面向对象代码的方式则可以看作是对应的已编译代码。Thomas 在其设计中提出了一种十分有价值的思路,即当复制一个闭包(closure)时,可以针对闭包的环境(environment)定制专门的复制函数,该函数会避免复制那些在特定函数中不会使用的环境变量。该策略不仅可以在复制环境变量时节省空间,更重要的是可以避免复制环境中已经不再使用的部分。Cheadle 等 [2004] 也针对每种闭包开发了专门的回收方法。Bartlett [1989a] 将这一思想应用在 C++ 的垃圾回收中,其所设计的回收器要求用户针对每个需要进行垃圾回收的类提供指针枚举方法。

在托管语言中,我们可以利用面向对象方法的间接调用过程实现特殊的回收相关操作。在 Cheadle 等 [2008] 的复制式回收器中,他们通过动态改变对象的函数指针来实现读屏障的自我删除(self-erase),这与 Cheadle 等 [2000] 在 Glasgow Haskell 编译器(GHC)中所使用的技术类似。该系统使用相似的技术实现了多种版本的栈屏障,除此之外还基于该技术实现

⊖ Bartlett [1998b] 在其 Scheme 实现中使用了这一方案(其实现方案是将 Scheme 转译成 C 代码),Cheadle 等 [2000] 也在无停顿 Haskell 中使用了这一方案。

了一个在更新待计算值 (thunk) 时使用的分代间写屏障。能够更新闭包环境的系统存在一项优势, 即它可以对现有对象进行收缩, 而为确保堆可解析性, 系统又需要在收缩完成之后在堆中插入一个伪对象。相应的, 系统也可能需要扩展某一对象, 此时系统便会用一个中转对象覆盖原有对象, 同时在中转对象中保存指向扩展对象的指针, 后续的回收过程也可以将中转对象优化掉。回收器也可以额外为赋值器执行一些计算, 例如对部分参数已经完成计算的“知名”函数提早执行计算, 返回链表首个元素这一函数[⊖]即为“知名”函数的一个例子。

170

从原则上讲, 静态类型语言可以省略对象头部并节省空间。Appel[1989] 和 Goldberg[1991] 描述了如何在 ML 语言中实现这一要求。在他们的解决方案中, 回收器只需要了解根的类型信息 (因为回收的追踪过程必须有一个起点)。但 Goldberg 和 Gloger[1992] 后来又发现, 回收器仍可能需要获取全部对象的类型信息, 这取决于程序所使用的多态类别, 具体可参见 [Goldberg, 1992]。

11.2.4 全局根中的精确指针查找

全局根中的精确指针查找相对来说较为简单, 在对象中查找指针的技术大多都可以在这里复用。在全局根这一方面, 不同语言之间的主要差别在于全局根集合是否可以动态增长。动态代码加载是导致全局根集增长的原因之一。某些系统在启动时便包含一个基本的对象集合, 例如, Smalltalk、某些 Lisp 以及某些 Java 系统在启动时 (特别是在交互式环境中启动时) 便会包含一个基本的系统“映像”, 也称为引导映像 (boot image), 其中包括众多的类系 / 函数及其对象实例。程序在执行过程中可能会对引导映像进行局部修改, 从而导致引导对象引用了运行时创建的新对象, 此时回收器就必须把这些引导对象中的域也当作程序的根。在程序的运行过程中, 引导对象也可能成为垃圾, 因此偶尔对引导映像进行追踪并找出其中的不可达对象也是一个不错的选择。是否需要关注引导映像通常取决于是否使用分代式回收策略, 此时我们可以将引导映像看作是特殊的年老代对象。

11.2.5 栈与寄存器中的精确指针查找

在栈中精确查找指针的一种解决方案是将活动记录分配在堆中, 正如 Appel[1987] 所建议的那样, [Appel and Shao, 1994, 1996] 也使用同样的方案, 且 Miller 和 Rozas [1994] 再次论证了该方案的可行性。某些语言实现使用与管理堆相同的方式来管理栈帧, 从而达到了一箭双雕的效果, 例如 Glasgow Haskell 编译器 [Cheadle 等, 2000] 以及 Non-Stop Haskell [Cheadle 等, 2004]。语言的实现者也可以专门为回收器提供一些关于栈上内容的相关指引, 例如 Henderson [2002] 在 Mercury 语言中便以这种方式来处理用户生成的 C 代码, Baker 等 [2009] 在为 Java 进行实时性改造时也使用了类似的技术。

但是, 出于多方面的效率因素, 大多数语言都会对栈帧进行特殊处理以获取最佳的运行性能, 此时回收器的实现者便需要考虑以下 3 个问题:

- 1) 如何在栈中查找帧 (活动记录);
- 2) 如何在帧中查找指针;
- 3) 如何处理以约定方式传递的参数、返回值, 以及寄存器中值的保存与恢复。

在大多数系统中, 需要在栈中查找帧的不仅仅只有回收器, 诸如异常处理与恢复等其他机制也需要对栈进行“解析”, 更不用说调试环境下至关重要的栈检查功能了。同时, 栈的

⊖ 即 Lisp 语言中的 car 函数。——译者注

可解析性也是某些系统（如 Smalltalk）自身的要求。从开发者的角度来看，栈本身当然是十分简洁的，但在这个简洁的外表背后，真正的栈在实现上却是经过高度优化的，帧的布局通常也更加原始。由于栈的可解析性通常十分有用，所以帧的布局管理通常需要支持这一点。

[171] 例如，在许多栈的设计实现中，每个帧中都会有一个域用于记录指向上一帧的动态链表指针，而其他各域均位于帧中固定偏移量的位置（此处的偏移量是相对于帧指针或者动态链表指针所指向的地址）。许多系统中还会包含一个从函数返回地址到其所在函数的映射表，在非垃圾回收系统中该表通常只是调试信息表的一部分，但许多托管系统却需要在运行时访问该表，因此该表就必须成为程序代码的一部分（可以在启动时加载到程序，也可以在程序启动后生成），而不能仅作为辅助调试信息来使用。

为确保回收器可以精确地找出帧中的指针，系统可能需要为每个栈显式增加栈映射（stack map）信息。这一元数据可以通过位图来实现，即通过位图中的位来记录帧中的哪些域包含指针。除此之外，系统也可以将帧划分为指针区和非指针区，此时元数据中所记录的便是两个区各自的大小。需要注意的是，当栈帧已经存在但尚未完全初始化时，函数可能会需要插入额外的初始化指令，否则回收器在这一状态下进行栈扫描则可能遇到问题。我们将在 11.6 节介绍安全回收点以及回收器与赋值器的握手，届时将会给出这一问题的解决方案。另外，我们可能需要对帧初始化代码进行回收方面的仔细分析，同时也必须谨慎地使用 push 指令（如果机器支持的话）或者其他特殊的压栈方式。当然，如果编译器可以将帧中的给定域固定当作指针或者非指针来使用，则帧扫描的实现便十分简单，此时所有的函数只需共享同一个映射表即可。

但是，单一栈映射方案通常不可行，如果使用该方案，则至少两种语言特性无法实现：

- 泛型 / 多态函数；
- Java 虚拟机的 jsr 指令。

我们曾经提到，多态函数可能会使用同一段代码来处理指针和非指针值，由于单映射表无法对两种情况进行区分，所以系统需要一些额外的信息。尽管多态函数的调用者可能“知道”具体的调用类型，但调用者本身也可能是一个多态函数，因而调用者需要将这一信息传递给更上层的调用者。因此在最差情况下，可能需要从 main() 函数开始逐级传递类型信息。这将与从根开始识别对象类型的策略十分类似 [Appel, 1989b ; Goldberg, 1991 ; Goldberg and Gloger, 1992]。

Java 虚拟机通过 jsr 指令来实现局部调用（local call），该指令不会创建新的帧，但它所调用的代码却能够以调用者的角色来访问当前帧中的局部变量。Java 通过该指令来实现 try-finally 特性，在正常以及异常逻辑下，finally 块中的代码都会通过 jsr 指令来调用。这里的问题在于，当虚拟机调用 jsr 指令时，某些局部变量的类型可能会出现歧义，此时局部变量的类型可能会取决于通过 jsr 指令调用 finally 块的调用者。对于某一未在 finally 块中使用但在未来会用到的变量，在正常调用逻辑下，它可能会包含指针，而在异常调用逻辑下，它又可能不包含指针。有两种策略可以解决这一问题。一种方案是依赖 jsr 指令的调用者来消除歧义，此时栈映射中域的栈槽类别就不能简单地划分为指针和非指针两种（即可以通过一个位来表示），还需要包含“询问 jsr 调用者”这第三种类别。此时我们需要找到 jsr 调用的返回地址，为达到这一目的，程序需要通过 Java 字节码进行一定的分析。另一种方案是简单地将 finally 块复制一份，虽然这可能改变字节码或者动态编译代码，但该方案在现代系统中应用得更加广泛。尽管该方案在最差情况下可能会造成代码体积指数级别的膨

胀，但它确实简化了系统中 `finally` 块的设计。据说有证据表明，为动态编译代码生成栈映射是某些隐蔽错误的重要来源，因而在这里控制系统的复杂度可能更加重要。某些系统会将栈映射的生成延迟到回收器真正需要它的时候，尽管这样可以节约正常执行逻辑下的时间和空间，但可能会增加回收停顿时间。

系统选用单一栈映射的另一个问题是：它会进一步限制寄存器的分配方式，即每个寄存器都只能固定存储指针或者非指针。因此这一因素便首先决定了单一栈映射方案不适用于寄存器数量较少的机器。

需要注意的是，不论我们是为每个函数创建一个栈映射，还是为一个函数的不同部分创建不同的栈映射，编译器都必须确保调用层次最深的函数也能获取栈槽的类型信息。如果我们在开发编译器之前就能意识到这一需求的重要性，则实现起来并不会特别困难，但是如果要对现有的编译器进行修改，则难度相当大。

寄存器中的指针查找。到目前为止，我们都忽略了寄存器中的指针。寄存器中的指针查找会比栈中的指针查找复杂得多，这是由以下几个原因决定的：

- 我们曾经提到，对于某个具体的函数，编译器可以固定地将其栈帧中的某个域用作指针域或者非指针域，但这一方案通常不能简单地套用在寄存器上，或者存在较大的局限性：该方案需要在寄存器中划分出两个特殊的子集，一个集合中的寄存器只能用作指针，而另一个只能用作非指针，因此此方案可能只适用于寄存器数量较多的机器。在大多数系统中，每个函数可能会对多个寄存器映射表。
- 即使我们可以确保所有全局根、堆中对象、局部变量都不包含内部指针（见 11.2.7 节）和派生指针（见 11.2.8 节），但经过高度优化的本地代码序列仍有可能导致寄存器持有这样的“非正规”指针。
- 函数调用约定（call convention）要求某些寄存器遵守调用者保存（caller-save）协议，即如果调用者想要在调用过程完成后继续使用某一寄存器中的值，其必须在发起调用之前将该寄存器的值保存下来；相应地，还有一些寄存器会遵守被调用者保存（callee-save）协议，即被调用者在使用某一寄存器之前必须先将其中的值保存下来，并确保在使用完成后将其恢复。对于寄存器中的指针查找，调用者保存寄存器（caller-save register）并不存在太大困难，因为调用者必然知道该寄存器所保存数据的类别，但被调用者保存寄存器中的值究竟是何种类别，则只有上层调用者（如果存在的话）才会知道。因此，被调用者无法确定一个尚未保存的被调用者寄存器是否包含指针，即使被调用者将该寄存器的值保存到帧的某一域中，同样无法确定该域是否包含指针。

许多系统都通过栈展开（stack unwinding）机制来实现栈帧以及调用链的重建（reconstruct），特别是对于没有提供专用的“上一帧”寄存器的系统。

下面我们将介绍一种寄存器指针查找策略，该策略可以解决被调用者保存寄存器中的指针查找问题。该策略首先要求为每个函数增加一个元数据，其中记录的信息包括该函数保存了哪些被调用者保存寄存器，以及每个寄存器的值保存在帧的哪个域中。我们假定系统所使用的是最常见的一种函数调用方案，即函数一开始便将其可能用到的被调用者保存寄存器保存到帧中。如果编译器较为复杂，以至于同一函数内的不同代码段都可能以不同的方式使用寄存器，则其需要为函数内不同的代码段分别插入被调用者保存寄存器的相关信息。

从顶层帧开始，我们重建寄存器时首先应当恢复被调用者保存寄存器，并获取这些寄存

器在调用者执行调用时的状态。为确保栈回溯顺利，我们需要记录哪些寄存器得到恢复，以及恢复操作后所获取到的值。到达调用栈底部的函数之后，所有的被调用者保存寄存器均可忽略（因为它不存在任何调用者），此时我们便可确定所有寄存器中的指针，回收器可以使用该信息并在必要时更新其中的值。

栈回溯过程需要恢复被调用者保存寄存器。需要注意的是，如果回收器更新了某一指针，则它同时也需要更新已保存的寄存器值。一旦函数使用了被调用者寄存器，我们便需要从额外的表中获取该寄存器原有的值，必要时，回收器还需要对该值进行更新。后续处理调用者的过程中，我们应当避免对已经处理过的被调用者保存寄存器进行二次处理。在某些回收器中，对根进行二次处理不会存在副作用（例如标记-清扫回收器），但在复制式回收器中，我们会很自然地认为所有尚未转发的引用都位于来源空间中，因此如果回收器两次处理相同的根（不是两个引用了同一对象的根），则可能会在目标空间中生成一份额外的副本。

算法 11.1 详细描述了上述处理流程，图 11.2 中展示了一个具体的处理实例。算法 11.1 中，*func* 是回收器用于扫描帧和寄存器的函数，它可以是算法 2.2（标记-清扫回收、标记-整理回收）中 *markFromRoots* 函数 **for each** 循环中的代码段，也可以是算法 4.2（复制式回收）中 *collect* 函数扫描根的循环中的代码段。

算法 11.1 通过栈回溯来处理被调用者保存寄存器

```

1 processStack(thread, func):
2     Regs ← getRegisters(thread)           /* 线程当前的寄存器状态 */
3     Done ← empty                          /* 所有寄存器都尚未完成处理 */
4     Top ← topFrame(thread)
5     processFrame(Top, Regs, Done, func)
6     setRegisters(thread, Regs) /* 将修正完成的寄存器值写回到线程状态中 */
7
8 processFrame(Frame, Regs, Done, func):
9     IP ← getIP(Frame)                     /* 当前指令指针 (IP) */
10    Caller ← getCalleeFrame(Frame)
11
12    if Caller ≠ null
13        Restore ← empty                   /* 保存在对调用者处理完成后应当恢复的值 */
14
15        /* 将 Regs 更新到调用者在发起调用之前的寄存器状态 */
16        for each {reg, slot} in calleeSavedRegs(IP)
17            add(Restore, {reg, Regs[reg]})
18            Regs[reg] ← getSlotContents(Frame, slot)
19        processFrame(Caller, Regs, Done, func)
20
21        /* 将被调用者保存寄存器更新后的值重新保存到其在帧内所对应的槽中 */
22        for each {reg, slot} in calleeSavedRegs(IP)
23            setSlotContents(Frame, slot, Regs[reg])
24
25        /* 从 Restore 中恢复 Reg，并调整 Done */
26        for each {reg, value} in Restore
27            Regs[reg] ← value
28            remove(Done, reg)
29
30        /* 处理当前帧的指针槽 */
31        for each slot in pointerSlots(IP)
32            func(getSlotAddress(Frame, slot))
33
34        /* 处理当前函数保存在寄存器中的指针 */
35        for each reg in pointerRegs(IP)

```

```
36      if reg ∉ Done
37          func(getAddress(Regs[reg]))
38          add(Done, reg)
```

我们首先来考虑图 11.2a 所示的调用栈（右侧带阴影的方框），其调用过程如下：

1) 程序从 `main()` 函数开始执行，初始状态下寄存器 `r1` 的值为 155，`r2` 为 784。为确保效率，`main()` 函数的调用者应当位于整个垃圾回收体系之外，因而其所对应的帧不得引用任何堆中对象，其寄存器的值也不能是指针。类似地，我们也无需关注 `main()` 函数的返回地址 `oldIP`。`main()` 函数所执行的操作依次是：将 `r1` 保存至槽 1 中、将变量 2 设置为指向对象 `p` 的指针、将变量 3 赋值为 75。然后 `main()` 函数将调用函数 `f()`，在执行调用之前 `r1` 的值为 `p`，`r2` 的值为 784，函数返回地址为 `main() + 52`。

2) 函数 `f()` 首先保存返回地址，然后再将 `r2` 保存在槽 1、将 `r1` 保存在槽 2、将变量 3 赋值为 -13、将变量 4 赋值为指向对象 `q` 的指针。然后函数 `f()` 将调用函数 `g()`，在执行调用之前 `r1` 的值是指向 `r` 的指针，`r2` 的值为 17，返回地址为 `f() + 178`。

3) 函数 `g()` 保存返回地址，将 `r2` 保存在槽 1、将变量 2 设置为对象 `r` 的引用、将变量 3 赋值为 -7、将变量 4 设置为指向对象 `s` 的指针。

图 11.2a 中，每个粗方框代表一个函数的帧，每个方框之上的寄存器值表示函数开始执行时寄存器的状态，位于方框之下的寄存器值表示其发起函数调用时寄存器值的状态。这些寄存器的值都应当在后续的栈展开过程中得到恢复。

假设函数 `g()` 在执行过程中触发了垃圾回收。

4) 垃圾回收过程发生在函数 `g()` 中的 `g() + 36` 位置，此时 `r1` 的值为指向 `r` 的指针，`r2` 的值为指向 `t` 的指针。我们假定此时指令指针（IP）以及各寄存器的值都已经保存在被挂起线程的某个数据结构中，或者保存在垃圾回收过程的某一帧中。

在某一时刻，回收器会在线程栈上调用 `processStack` 函数，参数 `func` 即为回收器扫描帧和寄存器的函数。对于复制式回收器而言，`func` 即 `copy` 函数，此时由于目标对象会发生移动，因而回收器需要更新栈以及寄存器中的引用。图 11.2a 左侧的方框展示了处理过程中变量 `Regs` 和 `Restore` 的变化，回收器将依照 `g()`、`f()`、`main()` 的顺序进行处理。我们对 `Regs` 和 `Restore` 的快照在左侧进行编号，编号的顺序与我们下面所描述的执行步骤保持一致。

5) `processStack` 函数将线程状态中的当前寄存器值写入 `Regs` 中，并将 `Restore` 初始化为空。此时函数执行到算法 11.1 的第 15 行，其所处理的帧为函数 `g()` 的帧。

6) 算法执行到第 19 行，处理的帧依然为函数 `g()` 所对应的帧。此时我们已经完成了 `Regs` 的更新，并且已经将函数 `g()` 在触发垃圾回收之前对寄存器的修改保存在了 `Restore` 中。由于函数 `g()` 一开始便将 `r2` 的值保存在槽 1，所以我们可以推断出在函数 `f()` 调用函数 `g()` 的时刻，`r2` 的值应当为 17。在函数 `g()` 发起垃圾回收的时刻，`r2` 的值为 `t`，我们将这一信息保存在 `Restore` 中[⊖]。在进一步对函数 `g()` 进行处理之前，我们先递归调用 `processStack` 函数对其调用者进行处理。图 11.2a 中，我们把 `calleeSavedRegs` 函数所返回的对组以及指令指针记录在函数 `g()` 所对应帧的左侧。

7) 算法再次执行到第 19 行，此时所处理是函数 `f()` 所对应的帧，我们从槽 2 和槽 1 中分别恢复 `r1` 和 `r2` 的值。

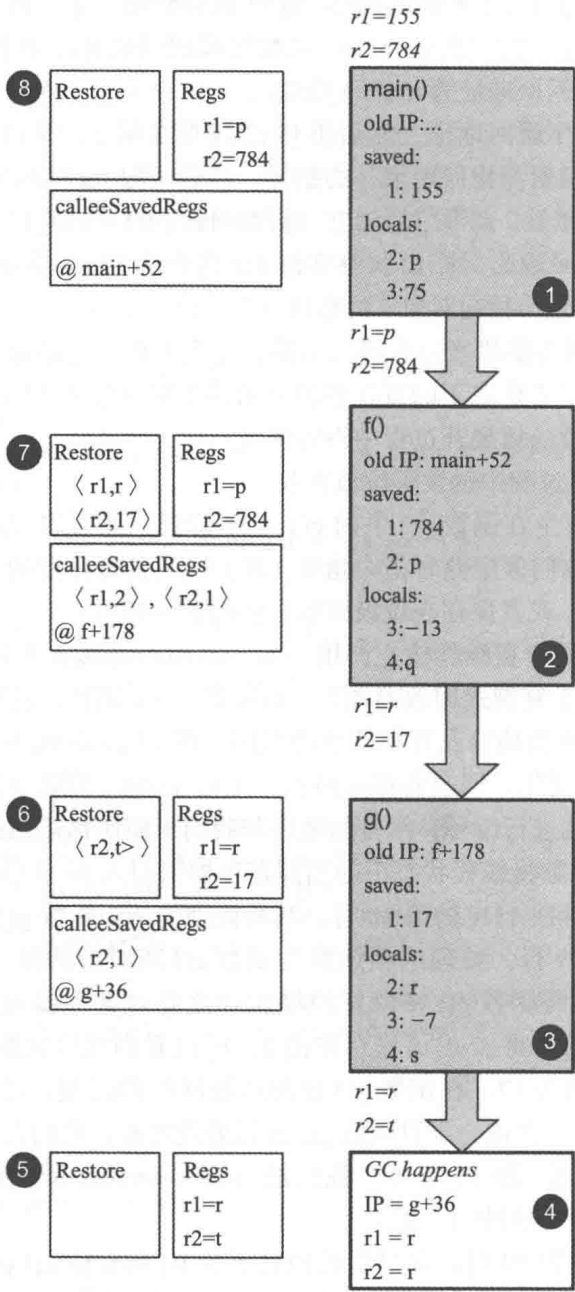
8) 算法再次执行到第 19 行来处理函数 `main()` 的帧。由于函数 `main()`“不存在”调用者，

⊖ 由于函数 `g()` 并未修改寄存器 `r1`，所以无需将其记录在 `Restore` 中。——译者注

174
175

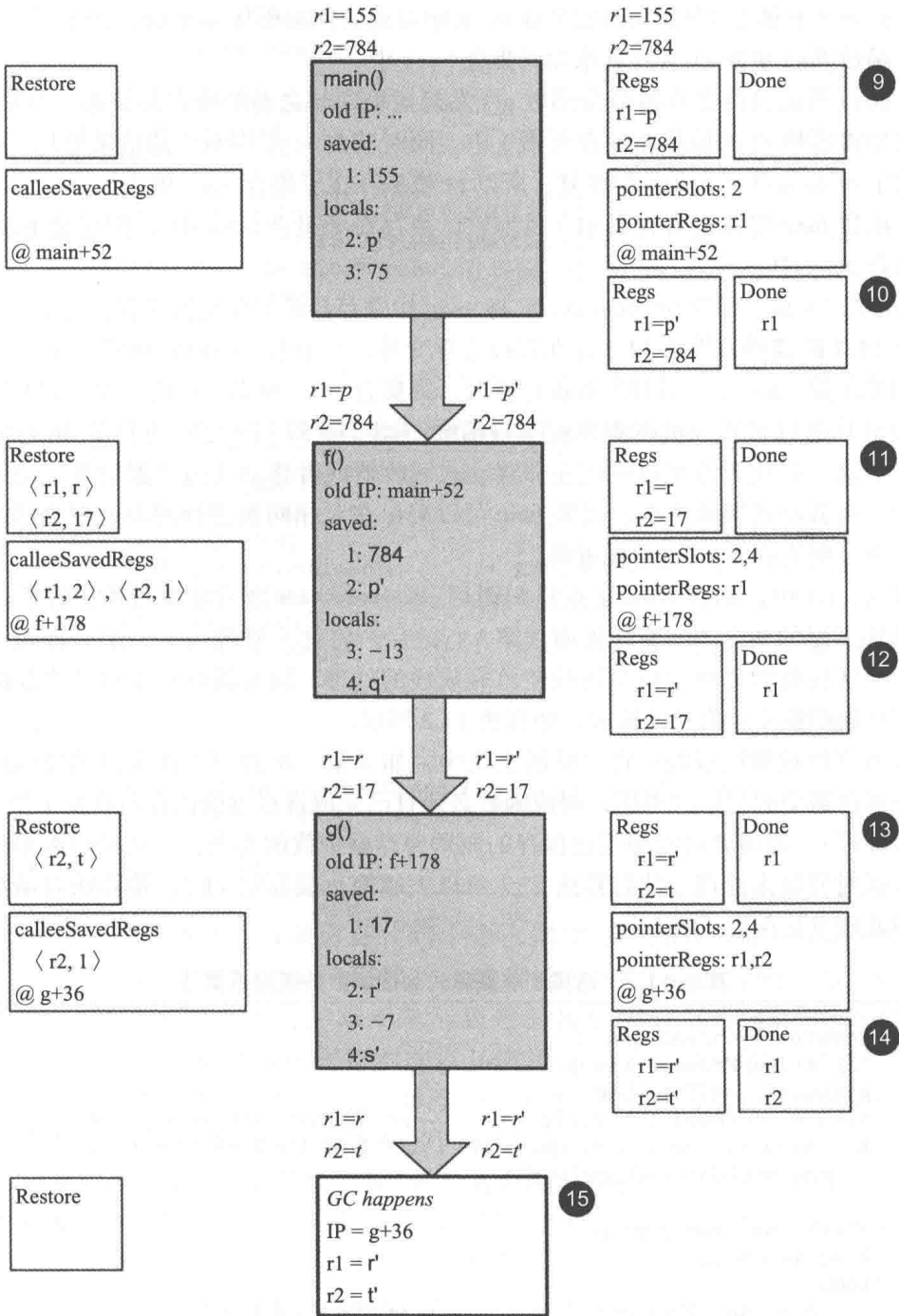
所以我们无需恢复任何的被调用者保存寄存器。更加确切地讲，应该是 `main()` 函数的调用者位于整个垃圾回收体系之外，其任何寄存器都不会包含与垃圾回收相关的指针。

完成函数 `main()` 在调用函数 `f()` 之前的寄存器数据重建之后，我们便可以对 `main()` 函数的帧以及寄存器进行处理，函数 `f()` 和 `g()` 也使用完全相同的方法来处理。接下来我们通过图 11.2b 来介绍每个帧将会到达的两种状态，一种状态对应算法 11.1 的第 35 行，另一种对应第 38 行之后。图 11.2b 所反映的是各个帧在算法第 35 行的状态，其中加粗的值表示已更新的值（尽管该值可能并不需要更新），灰色表示未更新的值。



a) 从栈顶开始回溯

图 11.2 栈扫描



b) 退回到栈顶

图 11.2 (续)

9) **Regs** 所记录的是函数 **main()** 调用函数 **f()** 之前的寄存器状态, 此时集合 **Done** 依然为空。

10) 函数 **func** 对寄存器 **r1** 进行更新 (因为 **r1** 属于 **main() + 52** 处的集合 **pointerRegs**), 并将其加入集合 **Done** 中, 目的是记录 **r1** 已经更新到其所引用对象的新地址 (如果存在的话)。

11) **Regs** 所记录的是函数 **f()** 调用函数 **g()** 之前的寄存器状态。注意, **r1** 和 **r2** 的值需要

重新保存到槽 1 和槽 2 中[⊖]，同时它们在 Regs 中对应的值需要从 Restore 中恢复[⊖]。

12) 函数 *func* 更新 r1 并将其添加到集合 Done 中。

13) Regs 所记录的寄存器值是函数 g() 发起垃圾回收之前的寄存器状态。与第 11 步类似，回收器需要将 r2 的值重新保存到槽 1 中，同时其在 Regs 中对应的值需要从 Restore 中恢复。由于 r1 并未从 Restore 中恢复，所以 r1 依然存在于集合 Done 中。

14) 函数 *func* 将跳过寄存器 r1 (因为它已经存在于集合 Done 中)，但它会更新 r2 并将其加入集合 Done 中。

最后，第 15 步，函数 processStack 将 Regs 中寄存器的值恢复到线程状态中。

算法 11.1 的变种。算法 11.1 存在多种合理变种，其中有一些值得我们关注：

- 如果函数 *func* 不会对其参数进行修改，则集合 Done 便无存在的必要，回收器既不需要对其进行更新，也不需要执行函数 *func* (第 37 行) 之前进行第 36 行的条件判断。这一简化策略可以应用在非移动式回收器或者移动式回收器的非移动处理过程中。在移动式回收器中，如果 *func* 可以确保在对相同根进行两次处理之后结果依然正确，则集合 Done 也可以省略。
- 算法 11.1 中，函数 *func* 是在递归调用 processFrame 之后才执行的，我们也可以将算法末尾的两个 for 循环提前到第 9 行之后。将这一修改与上一个变种相结合，新的算法便仅需要单方向遍历栈便可完成栈的处理，同时新算法也可以将栈的遍历方式从递归模式改为迭代模式，如算法 11.2 所示。
- 在不支持被调用者保存寄存器的系统中，如果某一函数希望在执行函数调用之后某一寄存器中的值依然可用，则该函数必须自己完成寄存器的保存与恢复工作 (即调用者保存)。调用者可以知道已保存的调用寄存器中值的类型，因而可以简单地将其当作临时变量来处理。此时算法 11.1 可以大幅简化成算法 11.3，该算法对函数调用栈的处理也是迭代式的。

算法 11.2 适用于非更新式处理函数的栈遍历算法

```

1 processStack(thread, func):
2   Top ← topFrame(thread)
3   processFrame(Top, func)
4   Regs ← getRegisters(thread)          /* 获取线程当前的寄存器状态 */
5   for each reg in pointerRegs(IP)      /* 在 GC 点从寄存器开始追踪 */
6     func(getAddress(Regs[reg]))
7
8 processFrame(Frame, func):
9   Done ← empty
10  loop
11    IP ← getIP(Frame)                  /* 当前指令指针 */
12
13    /* 处理当前帧的指针槽 */
14    for each slot in pointerSlots(IP)
15      func(getSlotAddress(Frame, slot))
16
17    /* 处理当前函数保存在寄存器中的指针 */
18    for each reg in pointerRegs(IP)
19      if reg ∉ Done
```

⊖ 算法第 23 行。——译者注

⊖ 算法第 27 行。——译者注

```

20      func(getAddress(Regs[reg]))
21      add(Done, reg)
22
23      Caller ← getCallerFrame(Frame)
24      if Caller = null
25          return
26
27      /* 将 Regs 更新到调用者发起调用时的寄存器状态 */
28      for each (reg,slot) in calleeSavedRegs(IP)
29          Regs[reg] ← getSlotContents(Frame, slot)
30          remove(Done, reg)
31
32      Frame ← Caller

```

算法 11.3 不存在被调用者保存寄存器时的栈遍历

```

1  processStack(thread, func):
2      Top ← topFrame(thread)
3      processFrame(Top, func)
4      Regs ← getRegisters(thread)          /* 线程未来的寄存器的状态 */
5      for each reg in pointerRegs(IP)      /* 在 GC 点从寄存器开始追踪 */
6          func(getAddress(Regs[reg]))
7      setRegisters(thread, Regs)          /* 将修正过的寄存器值写回到线程寄存器中 */
8
9  processFrame(Frame, func):
10     repeat
11         IP ← getIP(Frame)                /* 当前 IP */
12         for each slot in pointerSlots(IP) /* 处理帧上的指针槽 */
13             func(getSlotAddress(Frame, slot))
14         Frame ← getCallerFrame(Frame)
15     until Frame = null

```

栈映射压缩。经验表明，栈映射在代码中会占据相当一部分空间。Diwan 等 [1992] 发现，对于 VAX 平台上的 Modula-3 程序，栈映射会占据 16% 的代码空间；Stichnoth 等 [1999] 声称，对于 x86 平台上的 Java 应用程序，栈映射会占据 20% 的代码空间。Tarditi [2000] 提出了一种栈映射压缩技术并将其应用在 Marmot Java 编译器中，结果表明，栈映射的压缩率可以达到 4 ~ 5 : 1，最终让栈映射在代码所占空间的比例平均下降到 3.6%。该策略是以下面两个观察经验为基础的：

- 尽管程序中可能存在很多需要使用栈映射的回收点 (GC-points)，但许多回收点的栈映射都是完全相同的，因此可以通过多个回收点共享栈映射的方式来节省空间。这一特性在 Marmot 系统中尤为明显，因为在该系统中，经过回收点之后依然存活的指针数量通常较少。Tarditi [2000] 发现，仅凭借这一策略便可将栈映射的空间降低一半。
- 如果编译器可以将指针在帧内集中布局，则栈映射相同的比例会更高；通过存活变量分析与染色，编译器可以使生命周期不存在交集的变量复用相同的槽，也可增加栈映射相同的比例。Tarditi [2000] 发现对于大型程序而言这一策略尤为重要。

Tarditi 的整体处理流程如下：

1) 将返回地址 (稀疏) 集合映射到 (更小、更紧密的) 回收点编号集合[⊖]，即：如果 $t[i]$ 所对应的栈映射与返回地址 ra 相同，则将 ra 映射到回收点 i 。

178
179

[⊖] Tarditi 使用“调用点”(call site)这一术语，而我们使用的是“回收点”(GC-point)。

2) 将回收点编号映射到(较小的、较紧密的)栈映射编号集合。由于许多回收点的栈映射都相同,因而可以达到节省空间的目的。对于给定的回收点 i ,其所对应的栈映射编号为 $mn = \text{mapnum}[i]$ 。

3) 使用数组将栈映射编号映射到具体的栈映射信息。对于上一步中给定的 mn ,其所对应的栈映射信息为 $\text{info} = \text{map}[mn]$ 。

在 Tarditi 的方案中,栈映射信息是一个 32 位的字。如果 31 个位便足以表达栈映射信息,则该字的最低位将为零;否则该字的最低位为 1,而高 31 位则指向了变长的、完整的栈映射信息。具体的实现细节可能需要根据不同的目标平台(语言、编译器、处理器架构)进行调整,其实现代码可以参见作者的论文。

Tarditi 同时还介绍了几种将 IP(指令指针)映射到回收点编号的方式:

- 为栈映射相同的相邻回收点赋予相同的编号,Diwan 等[1992]也使用了这一技术。对于编号相同的回收点,该方案仅记录第一个回收点编号,其余回收点(返回地址大于第一个回收点,但小于映射表中下一个条目中记录的返回地址的回收点)都可以看作是与第一个回收点等价。
- 使用两级映射表来组织较大的回收点地址数组。该方案会将代码空间划分为 64KB 的块,并为每个代码块建立独立的栈映射表。由于每个代码块中所有回收点的高位都相同,所以表中的每一项只需要记录低 16 位,那么在 32 位地址空间中仅凭该策略便可节省一半的栈映射表空间。我们同时还要知道每个代码块中第一个回收点的全局编号。对于给定 IP,先获取其在代码块内部的回收点编号,然后再将该值与本代码块中第一个回收点的全局编号相加,便可得到其所对应回收点的全局编号。
- 使用稀疏回收点数组记录部分 IP 所对应的栈映射编号,而其他 IP 所对应的回收点编号则通过插值(interpolate)的方式得出。该方案首先粗略选定某一 IP 附近的 k 字节代码段,确定该代码段的位置、其中所有的回收点的编号及其所对应的栈映射信息编号。当需要确定某一 IP 所对应的回收点编号时,处理过程首先从该代码段中地址最高的 IP 位置开始向前进行代码反汇编,每遇到一个函数调用(或者其他回收点),便更新回收点编号以及栈映射编号。需要注意的是,处理过程必须能够检查出回收点的位置。Tarditi 发现,尽管这一过程需要使用 16 个元素缓存来减少相同返回地址的重复计算,但即使对于 x86 平台,这一反汇编过程也不是特别复杂的,执行过程也不算太慢。该策略的压缩率最高,且反汇编开销通常也不大。

Stichnot 等[1999]提出了另一种不同的栈映射压缩方案,该方案尝试为每条指令创建对应的栈映射。与 Tarditi[2000]所使用的稀疏数组类似,该方案仅为代码中某些固定的引用点(reference point)记录完整的栈映射信息,该回收点之后的 IP 所对应的栈映射信息则通过对引用点之后的代码进行反汇编得出。在 Stichnot 等人的方案中,引用点所对应的是真正的栈映射信息而非回收点编号。引用点(大致)开始于代码中基本块的起始位置。如果某块代码中最后一个栈映射与下一块代码中的第一个相同,则将两个代码块合并为更大的块。处理过程从每个引用点开始,对回收点的指令长度(x86 采用变长指令集)及其所对应栈映射的偏移量进行编码。例如,指令可能是在栈上压入或者弹出一个值,或者将指针加载到寄存器等。他们进一步对偏移量信息使用 Huffman 编码进行压缩。针对某一基准程序套件的测试结果表明,栈映射所占用的空间平均可以降低到全部代码空间的 22%。他们声称,即使对于寄存器数量较多的平台,该策略也不会表现得更差,因为处理器的寄存器数量越多,指令数量

通常也会增多。尽管他们（和 Tarditi[2000] 一样）使用反汇编策略来避免在大多数情况下记录指令长度，但需要记录的指令长度仍会带来明显的额外开销，而使用定长指令集的平台则不存在这一问题。该方案同时还需要使用额外的标记位来记录不允许执行垃圾回收的代码位置，例如写屏障代码序列中的代码。在定长指令集中，每条指令的长度完全相同（例如都是 4 字节），但 x86 指令集的平均长度可能会比前者少一半甚至更多。对于使用定长指令集的平台，Stichnot 等人的策略可以将大多数应用程序的栈映射大小控制在整个代码大小的 5% ~ 10%。

11.2.6 代码中的精确指针查找

程序代码中可能会内嵌堆中对象的引用，特别是那些允许运行时加载代码或者动态生成代码的托管运行时系统。即使是对于事先编译好的代码，其所引用的静态 / 全局数据仍有可能在程序启动时从刚刚完成初始化的堆中分配。代码中的精确指针查找存在以下几个难点：

- 从代码中分辨出嵌入其中的数据通常较为困难，甚至不可能。
- 对于“不合作”的编译器所生成的代码，几乎不可能将其中的非指针数据与可能指向堆中对象的指针进行区分。
- 当指针被嵌入到指令中时，指针本身可能会被割裂成为好几小段。MIPS 处理器将 32 位静态指针值加载到寄存器中通常需要使用 `load-upper-immediate` 指令，该指令首先将一个 16 位的立即数加载到 32 位寄存器的高 16 位并将低 16 位清零，然后再使用 `or-immediate` 指令将另一个 16 位的立即数加载到寄存器的低 16 位。其他指令集也可能会出现类似的代码序列。此处的指针值算是一种特殊的派生指针（见 11.2.8 节）。
- 内嵌指针值可能并非直接指向其目标对象，具体可以参见我们对内部指针（见 11.2.7 节）以及派生指针（见 11.2.8 节）的讨论。

某些情况下我们可以通过代码反汇编来找出内嵌指针，但如果每次回收都需要反汇编全部代码并处理其中的根，则可能引入巨大的开销。当然，由于程序不会修改这些内嵌指针，因此回收器可以缓存其位置以提高效率。

更加通用的解决方案是由编译器生成一个额外的表来记录内嵌指针在代码中的位置。

某些系统简单地禁用内嵌指针，从而避免了这一问题。使用这一策略可能存在的问题是，在不同目标架构、不同的编译策略以及不同的访问特征下，代码的性能可能会有所不同。

目标对象可移动的情况。如果内嵌指针的目标对象发生移动，则回收器必须更新内嵌指针。更新内嵌指针的困难之一在于，出于安全性或者保密性原因，程序代码段可能是只读的，因此回收器可能不得不临时修改代码区的保护策略（如果可能的话），但这一操作可能会引发较大的系统调用开销。另一种策略则是禁止内嵌指针引用可移动对象。更新内嵌指针的另一个困难之处在于，对内存中代码的修改通常并不会使代码在其他指令高速缓存（instruction cache）中的副本失效或者强制更新，为此可能要求所有处理器将受影响的指令高速缓存行失效。在某些机器中，回收器在将指令高速缓存行失效之后可能还需要执行一个特殊的同步指令，目的是确保未来的指令加载操作发生在失效操作之后。另外，在将指令高速缓存行失效之前，回收器可能还需要将被修改的数据高速缓存行强制刷新到内存中（其中所保存的是回收器所修改的代码），并且需要使用同步操作来确保这一操作执行完毕。此处的实现细节与具体的硬件架构相关。

代码可移动的情况。一种特殊的情况是回收器可能会移动程序代码。此时回收器不仅要考虑目标对象可移动情况下的所有问题，更要考虑对栈以及寄存器中所保存的返回地址的修

正，因为回收器可能已经移动了返回地址所在代码。另外，回收器必须将所有与代码新地址相关的指令高速缓存行失效，并且小心地执行上文列出的所有相关操作。更深层次的问题在于，如果连回收器自己的代码都是可移动的，那么处理起来将更加复杂。最后，在并发回收器中进行代码移动将是一件极为困难的任务，此时回收器要么必须挂起所有线程，要么只能采用更加复杂的方式，即先确保新老代码都可以被线程使用，然后在一段时间内将所有线程都迁移到新代码，最后在确保所有线程都迁移完成的前提下将老代码所占用的空间回收。

11.2.7 内部指针的处理

所谓内部指针，即指向对象内部某一地址，但其所指向地址并非对象的标准引用的指针。更加准确地讲，我们可以把对象看作是一组与其他对象不重叠的内存地址集合，而内部指针所指向的正是该集合中的某一地址。回顾图 7.2 我们可以发现，标准的对象可能并不会与其任何一个内部指针相等。另外，对象真正占据的空间也可能会比开发者可见数据所需的空间要大。例如，C 语言允许指针指向数组末尾之外的数据，但对于数组而言这依然是一个合法的内部引用。

在某些系统中，语言级别的对象可能是由数个不连续的内存片段组成（例如 Siebert[1999]），但在描述内部指针（以及派生指针）时，我们这里的“对象”仅仅是指位于一块连续内存之上的（语言级别）对象。

回收器在处理内部指针时遇到的主要问题是判定其究竟指向了哪个对象，即如何通过内部指针的值来反推出其目标对象的标准引用。可行的方案有以下几种：

- 使用一张表来记录每个对象的起始地址。系统可以通过一个数组来维护对象的起始地址，数组可以使用两级映射的方式进行组织，即类似于 Tarditi[2000] 记录代码中回收点时所使用的策略（参见 11.2 节）。另一种策略是使用位图，位图中每一位均对应堆中一个内存颗粒（即内存分配单位），同时将对象首地址所在内存颗粒对应的位设置为 1。该方案可能适用于所有的分配器以及回收器。
- 如果系统支持堆的可解析性（参见 7.6 节），则回收器可以通过堆扫描来确定内部指针所指向的地址究竟落在哪个对象内部。如果每次都从堆的起始地址开始查找未免开销过大，因此系统通常会为堆中每个 k 字节的内存块记录其内部首个（或者最后一个）对象的起始地址，为了方便和确保计算效率， k 通常是 2 的整数次幂。回收器便可根据这一信息在内部指针所指向的内存块中进行查找，必要情况下可能需要从上一个内存块开始查找。使用额外的表会引入空间开销，而堆解析又会引入时间开销，回收器需要在这两者之间进行适当的取舍。更加详细的讨论将在 11.8 节展开（此处用到的技术即 11.8 节所介绍的跨越映射——译者注）。
- 如果使用页簇分配策略，则回收器可以通过内部指针所指向的内存块的元数据来获取对象的大小，同时也可计算出目标地址在内存块中的偏移量（将目标地址与合适的掩码进行与操作，获取该地址的低位），根据对象的大小将偏移量向下圆整，便可得到对象的首地址。

我们假设对于任意一个内部指针，回收器都能计算出其目标对象的标准引用。当某一内部指针的目标对象发生移动时（例如在复制式回收器中），回收器必须同时更新该内部指针，并且确保其目标地址在新对象中的相对位置与移动之前完全一致。另外，系统也可能会将对象钉住（pin），我们将在 11.4 节详细讨论。

如果系统允许使用内部指针,则由此带来的主要问题是:对内部指针的处理需要花费额外的时间和空间。如果内部指针数量相对较少,且可以与正规指针(tidy pointer)(即指向对象标准引用位置的指针)进行区分,则处理内部指针的时间开销可能不会太大。但是,如果要彻底支持内部指针,则可能需要引入额外的表(尽管具体的回收器通常会包含一些必要的表或者元数据),进而增大了系统的空间开销,同时维护该表也会引入额外的时间开销。

代码中的返回地址是一种特殊的内部指针,尽管它们并没有什么特殊的处理难度,但基于多种原因,回收器在查找某个返回地址所对应的函数时,所用的表通常会不同于其他对象。

11.2.8 派生指针的处理

Diwan 等[1992]将派生指针定义为:对一个或者多个指针进行算数运算所得到的指针。内部指针是派生指针的一个特例,它可以表示成 $p + i$ 或者 $p + c$ 这种简单形式,其中 p 为指针, i 为动态计算出的整数偏移量, c 为静态常量。由于内部指针所指向的地址必然位于对象 p 所覆盖的内存地址中的一个,所以其处理起来相对简单,但派生指针的形式则可以更加一般化,例如:

- $\text{upper}_k(p)$ 或者 $\text{lower}_k(p)$, 即指针 p 的高 k 位或者低 k 位。
- $p \pm c$, 但计算出的地址位于对象 p 之外。
- $p - q$, 即两个对象之间的距离。

某些情况下,我们可以根据派生指针来反推正规指针(即指向标准引用地址的指针),例如派生指针 $p + c$ 且 c 为编译期确定的常量。我们通常都必须知道生成派生指针的基本表达式,尽管该表达式本身可能也是一个派生指针,但追根溯源,必然可以找到产生派生指针的正规指针。

在非移动式回收器中,回收器可以简单地将正规指针当作根进行处理。但需要注意的是,在垃圾回收时刻,即使派生指针依然存活,其目标对象的正规指针仍有可能被编译器的存活变量分析判定为死亡,因此编译器必须为每个派生指针保留至少一个正规指针,但 $p \pm c$ 这一情况属于例外,因为回收器通过一个编译期常量对派生指针进行调整,便可计算出其所对应的正规指针,该过程不需要依赖其他运行时数据。

183

在移动式回收器中,派生指针的处理则需要编译器的进一步支持:为了记录每个派生指针是从哪个地址计算得出的,以及如何重建派生指针,编译器需要对栈映射进行扩展。Diwan 等[1992]给出了处理形如 $\sum_i p_i - \sum_j q_j + E$ 的派生指针的通用解决方案,其中 p_i 和 q_j 是正规指针或者派生指针, E 是一个与指针无关的表达式(即使 p_i 或 q_j 发生移动,该表达式也不会受到任何影响)。其处理流程是:先在派生指针的基础上减去 p_i 然后加上 q_j ,进而计算出 E 的值,然后执行移动,最后再根据移动之后的 p'_i 和 q'_j 以及 E 计算出新的派生指针值。

Diwan 等[1992]指出,编译器的优化可能会给派生指针的处理带来一些额外的问题,包括死亡基准变量(dead base variables)、多派生指针指向相同的代码位置(导致回收器在处理某一派生指针时需要涉及更多的变量)、间接引用(变量的值被记录在引用链的某个中间位置)等。为支持派生指针,编译器有时需要减少对代码的优化,但其影响通常较小。在VAX平台的Modular-3程序中,处理派生指针所需的表通常会占到程序大小的15%。

11.3 对象表

基于赋值器性能以及空间开销的考虑,许多系统都使用直接指向对象的指针来表示引

用。一种更加通用的方案是为每个对象赋予一个唯一标识，并通过某种映射机制来定位其具体数据的地址。对于对象所占空间较大且可能较为持久，但底层硬件地址空间却相对较小的场景，这一技术具有一定的吸引力。本节我们关注的正是堆如何适应地址空间。在上述场景中，对象表（object table）是一种十分有用的解决方案，除此之外，对象表在许多其他系统中同样十分有用。对象表通常是一个较为密集的数组，其中的每个条目引用一个对象。对象表可以仅包含指向对象数据的指针，也可以包含其他额外的状态信息。为确保执行速度，对象的引用通常是其在对象表中的直接索引，或者指向其在对象表中对应条目的指针。如果使用直接索引，则回收器迁移对象表的工作便十分简单，但系统在访问具体对象时却必须先获取对象表的基址，然后再执行偏移，如果系统可以提供一个专门的寄存器来保存对象表的基址，则这一操作并不需要额外的指令。

对象表的一个显著优点在于其可以简化堆的整理，即当需要移动某一对象时，回收器可以简单地移动对象并更新其在对象表中的对应条目。为简化这一过程，对象内部应当隐含一个自引用域（或者指向其在对象表中对应条目的指针），据此，回收器便可通过对对象的数据快速找到其在对象表中的对应条目。在此基础上，标记-整理回收器可以采用传统的方式完成标记（需要通过对象表间接实现），然后简单地“挤出”垃圾对象，从而实现对象数据的滑动整理。回收器可以将对象表中的空闲条目以空闲链表的方式组织。需要注意的是，将对象的标记位置于其在对象表的对应条目中的效率更高，这可以在检测或者设置标记位时节省一次内存访问操作。额外的标记位图也具有类似的优点。还可以将对象的其他元数据置于对象表中，例如指向其类型及大小信息的引用。

对象表本身也可以进行整理，例如使用 3.1 节所描述的双指针算法。也可以在整理对象数据的同时整理对象表，此时只需要进行一次对象数据遍历便可同时实现对象数据和对象表的整理。

如果编程语言允许使用内部指针或者派生指针，则对象表策略可能会存在问题，甚至会成为障碍。类似地，对象表也很难处理从外部代码指向堆中对象的引用，这一问题我们将在 11.4 节详述。如果编程语言禁止内部指针，则不论是否使用对象表，语言的具体实现都不会因此受到任何语义上的影响，但是有一种语言特征或多或少都需要依赖对象表来保证其实效率，即 Smalltalk 的 `become:` 原语。该原语的作用是将两个对象的身份互换，如果使用对象表，则其实现起来相当简单，赋值器只需要将它们在对象表中的对应条目互换即可。如果没有对象表的支持，`become:` 操作就可能需要对整个堆进行扫描。但即使不使用对象表，谨慎地使用 `become:` 操作也是可以接受的（Smalltalk 通常使用 `become:` 操作来设置对象的新版本），毕竟直接引用的方式在大多数情况下都会比对象表更加高效。

11.4 来自外部代码的引用

某些语言或者系统允许托管环境之外的代码使用堆中分配的对象，一个典型的例子便是 Java 原生接口（Java Native Interface），它允许 C、C++ 或者其他语言所开发的代码访问 Java 堆中的对象。更加一般化地讲，几乎每种系统都需要支持输入/输出，这一过程几乎必然需要在操作系统和堆之间进行一定的数据交换。如果系统需要支持外部代码和数据引用托管堆中的对象，那么将存在两个难点。难点之一在于，如果某一对象从外部代码可达，那么回收器如何才能正确地将其当作存活对象，并确保在外部代码的访问结束之前不会将该对象回收。我们通常只需要在调用外部代码期间满足这一要求，因而可以在发起外部调用线程的栈

中保留指向该对象的存活引用。

但是,某些托管对象也可能被外部代码长期使用,其可达范围也可能超出最初发起外部调用的函数。基于这一原因,回收器通常会维护一个已注册对象表来记录此类对象。如果外部代码需要在当前调用完成之后继续使用某一对象,则其必须对该对象进行注册,同时当外部代码不再需要且未来也不会再使用该对象时,必须显式将其注销。回收器可以简单地将已注册对象表中的引用当作额外的根。

另一个难点在于外部代码如何才能确定对象的地址,当然该问题只会在移动式回收器中出现。某些实现接口会将具体对象与外部代码相隔离,后者只有借助于回收器所提供的渠道才能访问堆中对象。此类接口对移动式回收器的支持较好。回收器通常会将指针转化为句柄之后再交由外部代码使用,句柄中会包含堆中对象的真正引用,也可能包含其他一些托管数据。此处的句柄相当于是已注册对象表中的条目,同时也是回收的根。Java 原生接口即采用这种方式实现外部调用。需要注意的是,句柄与对象表中的条目十分类似。

句柄不仅可以作为托管堆和非托管世界之间的一道桥梁,而且可以更好地适应移动式回收器,但并非所有的外部访问都可以遵从这一访问协议,特别是操作系统调用。此时回收器就必须避免移动被外部代码所引用的对象。为此,回收器可能需要提供一个钉住接口,并提供钉住 (pin) 和解钉 (unpin) 操作,当某一对象被钉住时,回收器将不会移动该对象,同时也意味着该对象可达且不会被回收。

如果我们在分配对象时便知道该对象可能需要钉住,则可以直接将其分配到非移动空间中。文件流 I/O 缓冲区便是以这种方式进行分配的。但程序通常很难事先判断哪个对象未来需要钉住,因此某些语言支持 pin 和 unpin 函数以便开发者自主进行任何对象的钉住与解钉操作。

185

钉住操作在非移动式回收器中不会成为问题,但却会给移动式回收器造成一定不便,针对这一问题存在多种解决方案,每种方案各有优劣。

- 延迟回收,或者至少对包含被钉住对象的区域延迟回收。该方案实现简单,但却有可能在解钉之前耗尽内存。
- 如果应用程序需要钉住某一对象,且对象当前位于可移动区域中,则我们可以立即回收该对象所在的区域(以及其他必须同时回收的区域)并将其移动到非移动区域中。该策略适用于钉住操作不频繁的场景,同时也适用于将新生代存活对象提升到非移动式成熟空间的回收器(例如分代回收器)。
- 对回收器进行扩展以便在回收时不移动被钉住的对象,但这会增加回收器的复杂度并可能引入新的效率问题。

我们下面将以基本的非分代复制式回收器为例来考虑如何对移动式回收器进行扩展,从而支持钉住对象。为达到这一目的,回收器首先要能将已钉住对象与未钉住对象区分。回收器依然可以复制并转发未钉住对象,但对于被钉住对象,回收器只能追踪并更新其中指向被移动对象的指针,却不能移动该对象。回收器同时还必须记录其所发现的已钉住的对象,当完成所有存活对象的复制之后,回收器不能简单地释放整个来源空间,而是只能释放已钉住对象之间的空隙。此时回收所获得的不再是一块单独的、连续的空闲内存,而可能是数个较小的、不连续的空间集合,分配器可以将每段空间当作单独的顺序分配缓冲区来使用。已钉住对象不可避免地会造成内存碎片,但在未来的回收过程中,一旦被钉住的对象得到解钉,由此造成的碎片便可消除。正如我们在 10.3 节看到的,某些主体非移动式回收器

也会采用类似的方案,即在存活对象间隙进行顺序分配 [Dimpsey 等, 2000 ; Blackburn and McKinley, 2008]。

钉住对象给移动式回收器引入的另一个难点在于,即使对象已被钉住,回收器仍需对其扫描和更新,但在执行该操作的同时,外部代码可能也在访问该对象,进而导致竞争的出现。为此,回收器不仅要将被外部代码引用的对象钉住,同时还可能需要钉住其所引用的其他对象。同样地,如果外部代码从某一对象开始遍历其他对象,或者仅判断/复制对象的引用而不关心其内部数据,回收器仍需将其钉住。

编程语言自身的特性或其具体实现也可能会依赖对象的钉住机制。例如,如果编程语言允许将对象的域当作引用来传递,则栈中可能会出现指向对象内部域的引用。此时我们可以使用 11.2.7 节所描述的内部指针相关技术来移动包含被引用域的对象,但该技术的实现通常较为复杂,且正确处理内部指针的代码可能会难以维护。因此某些语言实现通常会简单地将此类对象钉住,这便要求回收器能够简单高效地判定出哪些对象包含直接被其他对象(或者根)引用的域。该方案可以轻易解决内部指针的处理问题,但却无法进一步拓展到更一般化的派生指针问题(参见 11.2.8 节)。

11.5 栈屏障

我们在 11.2 节中介绍了栈上指针的查找技术,但是该技术却要求在查找过程中将赋值器线程完全挂起,直到扫描过程完成为止。在赋值器线程执行的同时扫描其帧必然是不安全的,因此必须将线程挂起一段时间,或者由线程自身完成其栈的扫描(即线程调用一个自扫描子过程,相当于是自我停顿),我们将在第 11.6 节详细介绍对线程寄存器以及栈进行扫描的合适时机。回收器可以使用增量式栈扫描策略,但也可以使用栈屏障(stack barrier)技术进行主体并发扫描。该方案的基本原理是在线程返回(或者因抛出异常而展开)到某一帧时对线程进行劫持。假设我们在栈帧 F 上放置了屏障,然后回收器便可异步地处理 F 的调用者及其更高层次的调用者等,同时我们可以确保在异步扫描的过程中,线程不会将调用栈退回到栈帧 F 中。

引入栈屏障的关键步骤在于劫持帧的返回地址,即将帧上保存的返回地址改写为栈屏障处理函数的入口地址,同时将原有的返回地址保存在栈屏障处理函数可以访问到的标准地址,例如线程本地存储中。栈屏障处理函数可以在合适的时候移除栈屏障,同时还应当小心确保不会对上层调用者的寄存器造成任何影响。

线程在对自身进行栈扫描时也可以使用增量扫描策略,即当赋值器线程陷入栈屏障处理函数时,其会向上扫描数个栈帧,并在扫描结束的位置设置新的栈屏障(除非处理函数已经完成整个栈的扫描)。我们将这一技术称为同步(synchronous)增量扫描。与之对应的,异步(asynchronous)增量扫描是由回收线程执行的,此时栈屏障的目的是在被扫描线程触达被扫描栈帧之前将其挂起。扫描线程在完成数个帧的扫描之后可以沿着调用栈的回退方向移动栈屏障,因此被扫描线程可能永远都不会触达栈屏障,一旦触达,则被扫描线程必须等待扫描线程执行完毕并解除栈屏障,然后才能继续执行。

Cheng 和 Blleloch [2001] 使用栈屏障技术来限制一个回收增量内的工作量,并借助该技术来实现异步栈扫描。他们将线程栈划分为固定大小的子栈(stacklet),每个子栈都可以一次性完成扫描,从一个子栈返回另一个子栈的位置即为栈屏障的备选位置。该方案并不要求各子栈连续布局,同时也不需要事先确定哪些帧上可以放置栈屏障。

回收器也能以另一种完全不同的方式来使用栈屏障，即利用栈屏障来记录栈中的哪些部分未改变过，因此回收器便不用每次都在这些位置中寻找新的指针。在主体并发回收器中，该技术可以减少回收周期结束时的翻转（flip）时间。

栈屏障的另一种用途是处理代码的动态变更，特别是经过优化的代码。例如，假设在某一场景下子过程 A 调用了 B，B 又调用了 C，我们进一步假定系统将 A 和 B 内联，即 A + B 共用一帧。如果用户修改了 B，则后续对 B 的调用应当执行到其新版本的代码中。因此，当线程从 C 中返回时，系统需要对 A + B 进行逆优化（deoptimise），同时分别为 A 和 B 的未优化版本创建新的帧，只有当线程从 B 返回到 A 之后，子过程 A 才能访问新版的子过程 B。系统甚至有可能重新进行优化并构建出新版的 A + B。此处我们关注的是，从 C 返回到 A + B 的过程将触发逆优化，而栈屏障正是触发机制的一种实现方案。

11.6 安全回收点以及赋值器的挂起

我们在第 11.2 节提到，回收器需要知道哪些栈槽以及哪些寄存器包含指针；我们同时还提到，如果垃圾回收发生在同一函数的不同位置（即 IP，也就是指令指针），这一信息通常会发生变化。对于哪些位置可以进行垃圾回收，有两个问题需要关注：第一，回收器是否可以在某一 IP 处安全执行垃圾回收；第二，如何控制栈映射表的大小（参见 11.2 节关于栈映射压缩的细节），如果允许在更多的位置执行垃圾回收，那么通常需要更大的栈映射表。

[187]

下面我们来考虑哪些原因可能导致回收器无法在某一 IP 处安全地进行垃圾回收。大多数系统通常都会存在一些必须作为整体来执行的短小代码序列，其目的在于确保垃圾回收需要依赖的一些不变式得到满足。例如，典型的写屏障不仅要执行底层写操作，还要记录一些额外的信息。如果垃圾回收过程发生在这两个阶段之间，则可能导致某些对象发生遗漏，或者某些指针被错误地更新。系统通常都会包含许多此类短代码序列，在垃圾回收器看来它们均应当是原子化的（尽管在严格的并发意义上讲它们并非真正的原子化操作）。更多的例子还包括新栈帧的创建、新对象的初始化等。

系统可以简单地允许回收器在任意 IP 位置发起垃圾回收，此时回收器将无需关心赋值器线程是否已经挂起在可以安全进行垃圾回收的位置，即安全回收点（GC-safe point）或者简称回收点（GC-point），但此类系统在实现上通常更加复杂，因为系统必须为每个 IP 提供对应的栈映射，或者只能使用不需要栈映射的技术（例如面向“不合作”的 C 和 C++ 编译器的相关技术）。假定系统允许回收器在绝大多数 IP 位置发起垃圾回收，那么如果某一线程在回收发起时挂起在不安全的 IP 位置，则回收器可以对线程挂起位置之后、下一个安全回收点之前的指令进行解析，或者将线程唤起一小段时间，以便其（在一定概率上可以）运行到安全回收点。指令解析会增加出错的风险，而将线程向前驱动一小段则只能在一定概率上保证其到达安全回收点。除此之外，此类系统所需的栈映射空间也可能会很大 [Stichnoth 等, 1999]。

许多系统使用另一种完全不同的策略，即只允许垃圾回收发生在特定的、已注册的安全回收点，同时也只为这些回收点生成栈映射。出于回收正确性的考虑，安全回收点的最小集合应当包括每个内存分配位置（因为垃圾回收通常会在此处发生）[⊖]、所有可能发生对象分配的子过程调用、所有可能导致线程挂起的子过程调用（因为在某一线程被挂起的同时，其他

[⊖] 对于线程在执行本地分配之前检查本地空闲空间是否足够的操作，系统可以不把它当作安全回收点。

线程有可能引发垃圾回收)。

为确保线程能够在有限时间内到达安全回收点,系统可以在安全回收点最小集合之外的更多位置增加回收点。为此系统可能需要在每个循环中增加安全回收点,一个简单的规则是将函数内部所有的后退分支设置为安全回收点。除此之外,系统也有必要在每个函数的入口以及返回位置设置安全回收点,否则线程在到达安全回收点之前可能需要经过大量函数调用,特别是递归调用。由于这些额外的回收点并不会真正触发垃圾回收,所以在线程这些位置只需要检查是否有其他线程发起垃圾回收,因此我们可以称其为回收检查点(GC-check points)。尽管回收检查点会给赋值器带来一定开销,但这一开销通常不大,编译器也可以通过一些简单的方法来减轻这一开销,例如当函数十分短小,或者其内部不包含循环或进一步函数调用时将回收检查点优化掉。为避免在循环的每次迭代中都执行回收检查,编译器也可以额外引入一层循环,即在每 n 轮迭代之后才执行回收检查。当然,如果回收检查的开销很小,这些优化手段便不再必要。总之,系统必须在回收检查的频率和回收发起时延之间做出平衡。

[188]

Agesen [1998] 对两种将线程挂起在安全回收点的策略进行了比较。一种策略是轮询(poll),即我们刚刚介绍的方案,该方案要求线程在每个回收检查点都要对一个旗标进行检查,该旗标被设置则意味着其他线程已经发起了垃圾回收。另一种方案是使用补丁(patching)技术,即当某一线程处于挂起状态时修改其执行路径上的下一个(或者多个)回收点的代码,线程恢复执行后便可在下一个回收点停顿下来。这与调试器在程序中放置临时断点的技术类似。Agesen 发现,补丁技术的开销要比轮询技术低得多,但其实现起来也更加复杂,在并发系统中也更容易出现问题。

在引出回收检查点这一思想时,我们曾经提到过回收器和赋值器之间的握手(handshake)机制。即使对于多个赋值器线程执行在相同处理器上的这种并非真正“并发”的情况,握手机制也是十分必要的,在回收启动之前,回收器必须将所有已经挂起、但挂起位置并非安全回收点的线程唤醒,并使其运行到安全回收点。为避免这一额外的复杂度,某些系统能够保证线程仅会在安全回收点挂起,但基于其他原因,系统可能无法控制线程调度的所有方面,因而仍需借助于握手机制。

下面我们将具体介绍几种特殊的握手机制。每个线程可以维护一个线程本地变量,该变量用于反映系统中的其他线程是否需要该线程在安全回收点关注某一事件。这一机制可以用于包括发起垃圾回收信号在内的多种场景。线程会在回收检查点检查这一本地变量,如果该变量非零,则线程会根据该变量的值执行具体的系统子过程。某个特殊的值将意味着“是时候进行垃圾回收了”,当线程发现这一请求之后,会设置另一个本地局部变量,该变量表示该线程已经准备就绪,除此之外线程也可以对某一回收器正在监听的全局变量执行自减操作来达到这一目的。系统通常会尽量降低线程本地变量的访问开销,因而该策略可能是一个不错的握手机制实现方案。

另一种方案是在被挂起线程已保存的线程状态中设置处理器条件码(processor condition code),因此线程在回收检查点便可通过一个十分廉价的条件分支来调用该条件码对应的系统子过程。该方案仅适用于包含多条件码集合的处理器(如 PowerPC),同时还必须确保线程在被唤醒之后不会处于外部代码的上下文中[⊖]。如果处理器的寄存器足够多,则可以使用

⊖ 外部代码可能正在使用这一条件码,因而回收器不能贸然进行修改。——译者注

一个寄存器来表示信号，而寄存器的使用开销几乎与条件码一样小。如果线程正在执行外部代码，则系统便需要通过某种方式来关注线程何时从外部代码返回（除非线程恰好被挂起在与安全回收点等价的位置），对返回地址进行劫持（也可参见第 11.5 节）是捕获线程从外部代码返回的策略之一。

除了设置旗标，外加返回地址劫持这一方案之外，系统还可以使用操作系统级别的线程间信号来实现握手，例如 POSIX 线程中的某些实现。该策略可能不具有广泛的可移植性，其执行效率也可能成为问题。影响效率的原因之一是信号传递的到用户级别处理函数需要通过操作系统内核级别的通道，而这一通道的处理路径相对较长。除此之外，这一机制不仅需要借助于底层处理器中断，还会影响高速缓存以及转译后备缓冲区，这也是影响其执行效率的重要原因之一。

综上所述，回收器与赋值器线程之间的握手机制主要有两种实现方式：一种是同步通知，也可称之为轮询，另一种是通过某种信号或者中断实现异步通知。每种实现机制都有相应的实现开销，而且具体的开销会随着平台的不同有所差异。轮询可能也需要编译器级别的协作，这取决于所选择的实现技术。另外，由于栈扫描等操作并非在任意时间都可以执行，所以异步通知通常需要转换成同步通知，此时信号处理函数的主要目的将是设置某一线程本地旗标，而线程则应当在发现该旗标被设置后做出响应。

我们需要进一步指出的是，如果各线程直接完成其栈的扫描，必须还要考虑硬件和软件层面的并发情况，此处可能会涉及第 13 章的相关内容。其中与握手机制相关性最大的内容可能是第 13.7 节，届时我们将介绍相关线程如何从回收的一个阶段迁移到另一个阶段，以及赋值器线程在回收的开始和结束阶段应当执行哪些工作。

189

11.7 针对代码的回收

许多系统会预先对所有代码进行静态编译，但也有一些程序可以在运行时构建并执行代码，例如垃圾回收技术的鼻祖——Lisp 语言。尽管 Lisp 最初是解释型语言，但其在很早就已经能够编译成本地代码。目前，越来越多的系统已经能够动态加载或者构建代码，并在运行时进行优化。由于系统可以动态加载或者生成代码，所以我们自然会希望当这些代码不再使用时，其所占用的空间能够得到回收。面对这一问题，直接的追踪式或引用计数算法通常无法满足该要求，因为许多从全局变量或者符号表可达的函数代码将永远无法清空。某些语言只能靠开发者显式卸载这些代码实例，但语言本身甚至可能根本不支持这一操作。

另外，还有两个特殊场景值得进一步关注。首先是由一个函数和一组环境变量绑定而成的闭包。我们假设某一简单的闭包是由内嵌在函数 f 中的函数 g ，以及函数 f 的完整环境变量构成的，它们之间可能会共享某一环境对象。Thomas 和 Jones[1994] 描述了一种系统，该系统可以在进行垃圾回收时将闭包的环境变量特化为仅由函数 g 使用的变量。该策略可以确保某些其他闭包最终不可达并得到回收。

另一种场景出现在基于类的系统中，例如 Java。此类系统中的对象实例通常会引用其所属类型的信息。系统通常会将类型信息及其方法所对应的代码保存在非移动的、不会进行垃圾回收的区域，因此回收器便可忽略掉所有对象中指向类型信息的指针。但是如果要回收类型信息，回收器就必须要对所有对象中指向类型信息的指针进行追踪，在正常情况下这一操作可能会显著增大回收开销。回收器可以仅在特殊模式下才对指向类型信息的指针进行追踪。

对于 Java 而言,运行时类是由其类代码以及类加载器(class loader)共同决定的[⊖]。由于系统在加载类时通常会存在一些副作用(例如初始化静态变量),所以类的卸载会变得不透明(即存在副作用——译者注),这是因为该类可能会被同一个类加载器重新加载。唯一可以确保该类不被某个类加载器加载的方法是使类加载器本身也能得到回收。类加载器中包含一个已加载类表(以避免重复加载或者重复初始化等),运行时类也需要引用其类加载器(作为自身标识的一部分)。因此,如果要回收一个类,则必须确保其类加载器、该类加载器所加载的其他类、所有由该类加载器所加载的类的实例都不被现有的线程以及全局变量所引用(此处的全局变量应当是由其他类加载器加载的类的实例)。另外,由于引导类加载器(bootstrap class loader)永远不会被回收,所以其所加载的任何类都无法得到回收。由于 Java 类卸载是一种特殊的情况,所以某些依赖这一特性的程序或者服务器可能会因此耗尽空间。

即使对于用户可见的代码元素(例如方法、函数、闭包等),系统也可能为其生成多份实例以用于解析或者在本地执行,例如经过优化的和未经优化的版本、函数的特化版本等。为函数生成新版本实例可能会导致其老版本实例在未来的调用中不可达,但这些老版本实例可能仍在当前的执行过程中得到调用,它们在栈槽或者闭包中的返回地址会保持其可达性。因此在任何情况下,系统都不能立即回收老版本代码实例,而只能通过追踪或者引用计数的方法来将其回收。此处的相关技术是栈上替换(on-stack replacement)技术,即系统使用新版本代码实例来替换其正在执行的老版本实例。该方案不仅可以提升正在运行的方法调用的性能,而且有助于回收老版本的代码,因而其使用越来越广泛,具体可参见 Fink、Qian[2003]以及 Soman、Krintz[2006]的栈上替换方案及其在 Java 中的应用。栈上替换技术的直接目的通常是优化代码或其他一些应用,例如需要对代码进行逆优化的调试需求,而在另一方面,回收器也可以利用该技术来回收老版本代码。

190

11.8 读写屏障

许多垃圾回收算法需要赋值器在运行时探测并记录回收相关指针(interesting pointer)。如果回收器仅回收堆中一部分区域,则任何从该区域之外指向该区域的指针都属于回收相关指针,且回收器必须在后续处理过程中将它们当作根。例如,分代垃圾回收器必须捕获所有将年轻代对象的引用写入年老代对象的写操作。我们将在第 15 章看到,当赋值器和回收器交替执行时(不论回收器是否运行在单独的回收线程之上),将很有可能出现赋值器操作导致回收器无法追踪到某些可达对象的情况,如果这些引用没有被正确地探测到并传递给回收器,则存活对象可能会被过早地回收。这些场景都要求赋值器即时地将回收相关指针添加到回收器的工作列表中,而这一任务的完成就需要借助于读写屏障。

在其他章节中,我们关注的主要都是特定的回收算法,读写屏障只是作为算法所必需的一部分被提及,但在本节,我们将对如何实现屏障这一问题进行总体介绍。本节我们将把各种特定回收算法(例如分代回收器或并发回收器)中的读写屏障进行抽象,并将注意力集中在回收相关指针的探测与记录上。探测(detection)即确定某一指针是否属于回收相关指针,而记录(record)则是对回收相关指针进行登记,以便回收器后续使用。探测和记录在某种程度上是正交的,但某些探测方法的使用可能会加强特定记录方法的优势,例如,如果写屏障通过页保护违例来进行探测,则对被修改位置进行记录会更加合理。

⊖ Java 虚拟机在判定两个 Java 类是否相同时,不仅要判断类的全名是否相同,还要判断加载此类的类加载器是否一样。——译者注

11.8.1 读写屏障的设计工程学

除了要执行真正的读/写操作之外,典型的屏障通常还会包括一些额外的检查与操作。典型的检查包括判断被写入的指针是否为空、被引用对象与其引用者所处分代之间的关系等,而典型的操作则是将对象记录到记忆集中。完整的检查以及记录操作可能太大,以至于无法整体内联,但这取决于屏障的具体实现。即使得到内联的指令序列相对短小,仍可能导致编译器生成的代码剧烈膨胀并进一步影响指令高速缓存的性能。由于屏障内部的大部分代码通常很少执行,所以设计者可以将指令序列划分为“快速路径”和“慢速路径”:快速路径通常会进行内联以确保性能,而慢速路径则只有在必要时才会调用,也就是说,为节省空间以及提升指令高速缓存的性能,慢速路径通常只会存在一份代码实例。快速路径应当包含最一般的情况,而慢速路径则仅应当在部分情况下执行,这一点十分重要。某些情况下,这一规则同样也适用于慢速路径的设计:如果屏障会经常进行多重检查,则设计者有必要对检查逻辑进行合理排序,并确保第一重检查会过滤掉大多数情况,第二重检查可以过滤掉次多的情况,以此类推,从而最大限度地降低检查开销。为达到这一要求,设计者通常需要对检查逻辑以多种方式进行排序并分别测量其性能,因为现代硬件环境中存在非常多的影响因素,以至于用简单的分析模型通常无法给出足够好的指引。

提升读写屏障性能的另一个因素是加速所有必需数据结构的访问速度,例如卡表。系统甚至可以付出一个寄存器的代价来保存某一数据结构的指针,例如卡表的基地址等,但是否值得如此取决于机器以及算法的类型。

设计者还需要对软件工程学有所关注,包括如何对垃圾回收算法的各个方面(即读写屏障、回收检查、分配顺序等)进行整合,它们都会被构建到系统的编译器中。如果有可能,设计者最好能为编译器指明哪些子过程需要内联,这些子过程内部应当是快速路径所对应的代码序列。这样一来,编译器便无需知道具体细节,而设计者则可以自由替换这些内联子过程。但正如我们前面所提到的,这些代码序列可能会存在一些限制,例如在其执行过程中不允许发生垃圾回收,这便需要设计者小心对待。编译器可能也需要避免对这些代码序列进行优化,例如保留一些显而易见的无用写入(它们所写入的数据对回收器有用)、禁止对屏障代码进行指令重排序或者与周围代码进行穿插。最后,编译器可能需要支持一些特殊的编译指示(pragma),或者允许设计者使用特殊的编译属性,例如不可中断的代码序列。

[191]

我们将在本节的剩余部分讨论写屏障。读屏障一般用于并发回收器和增量回收器中,而我们将其推迟到相关章节中介绍。写屏障比读屏障复杂得多,它不仅需要探测回收相关指针,而且还需要记录一些信息以供回收器后续使用。相比之下,读屏障通常只会触发一些立即性的操作,例如将刚刚加载的引用的目标对象复制到目标空间中。

11.8.2 写屏障的精度

回收相关指针的记录存在多种不同的实现策略与机制,具体的实现策略决定了记忆集记录回收相关指针位置的精度。在选择回收相关指针的记录策略时,我们需要对赋值器与回收器各自的开销进行平衡。实践中我们通常倾向于增加相对不频繁的回收过程(例如查找根集合)的开销,同时降低更为频繁的赋值器行为(例如堆的写操作)的开销。如果抛开写屏障的影响,指针的写入操作通常都很快(尽管托管代码通常会对空指针或者数组边界进行检查)。在引入写屏障之后,指针写操作所需的指令数可能会增大两倍或者更多,但如果写屏障的局部性比赋值器自身的局部性要好,则这一开销很可能会被掩盖(例如,写屏障在记录

回收相关指针时通常不会导致用户代码的延迟)。一般来说,记忆集中回收相关指针的记录精度越高,回收器查找操作的开销就会越低,而赋值器过滤并记录指针的开销则会越高。作为一种极端情况,分代式回收器中的赋值器可以不记录任何指针写操作,从而将所有回收相关开销转移给回收器,此时后者就只能扫描整个堆空间并找出所有指向定罪分代的引用。虽然这并非一种通用的较为成功的分代策略,但是对于无法借助于编译器或者操作系统的支持来捕获指针写操作的场景,这可能是唯一可选的分代策略,此时回收器可以使用局部性更好的线性扫描而非追踪策略来查找回收相关指针 [Bartlett, 1989a]。Swanson [1986] 和 Shaw [1988] 声称,与简单的半区复制策略相比,这一策略可以降低 2/3 的垃圾回收开销。

记忆集的设计策略需要从三个维度进行考虑。第一,回收相关指针的记录应达到何种精度。尽管并非所有的指针都是回收相关指针,但对于赋值器而言,无条件记录的开销显然会低于对回收无关指针执行过滤之后再记录。记忆集的具体实现是决定过滤开销的关键:如果记忆集可以使用非常廉价的机制来增加条目,例如简单地在某一固定大小的表中写入一个字节,则该策略非常适合无条件记录,特别是在添加操作本身满足幂等要求的情况下;另一方面,如果向记忆集添加条目的开销较高,或者记忆集的大小也需要控制,则写屏障过滤掉回收无关指针则显得十分必要。对于并发回收器或者增量回收器而言,过滤操作是必不可少的,只有这样才能确保回收器的工作列表可以最终为空。每种过滤策略都需要考虑过滤逻辑应当内联到何种程度,何时应当通过外部调用来执行过滤或将指针添加到记忆集。内联的指令越多,则需要执行的指令越少,但这可能导致代码体积的膨胀并增大指令高速缓存不命中的几率,进而影响程序的性能。因此,开发者需要对过滤检查的顺序以及需要内联的过滤操作进行精细化调节。

192

第二,对指针位置的记录应当达到何种粒度。最精确的方案是记录指针所写入的域的地址,但如果某一对象中的指针域较多(例如对数组进行更新时),则可能会增大记忆集的大小。另一种方案是记录被修改指针域所在的对象,其优势在于可以根据对象进行去重,对指针域进行记录通常无法做到这一点(因为在指针域中通常不会有额外的空间来标记该域是否已被记录)。记录对象的方法要求回收器在追踪阶段扫描对象内部的每个指针域,进而才能找到它们所引用的尚未得到追踪的对象。一种混合式解决方案是以对象为粒度来记录数组,而以指针域为粒度来记录纯对象,因为当数组中的一个域得到更新时,其他域通常也会得到更新。也可使用完全相反的策略,即以指针域为粒度来记录数组(以避免对整个数组进行扫描),而以对象为粒度来记录纯对象(纯对象通常比较小)。对于数组而言,还可以只记录数组的一部分,这一策略与卡标记(card mark)策略十分类似,不同之处在于其依照数组索引而非数组域在虚拟内存中的地址进行对齐。究竟应当记录对象还是记录域,还取决于赋值器可以获取到哪些信息:如果写操作既可以获取对象的地址又可以获取指针域的地址,则其可以任意选择一种;但如果写屏障只能获取被写入域的地址,则计算其所属对象的地址可能会引入额外的开销。Hosking 等 [1992] 在某一解释型 Smalltalk 系统中解决了这一难题,它们的策略是在顺序存储缓冲区中同时记录对象以及域的地址。

卡表(card table)技术(我们将在稍后介绍)将堆在逻辑上划分为较小且固定大小的卡。该方案以卡为粒度来记录指针的修改操作,其记录方式通常是在卡表中设置一个标记字节。卡标记不仅可以对应被修改的域,也可以对应被修改的对象(两类信息可以对应不同的卡)。在回收阶段,回收器必须先找到所有与待回收分代相关的脏卡,然后找出其中记录的所有回收相关指针。卡表的记录方式(记录对象还是记录域)会影响查找过程的性能。比卡表的粒

度更粗的记录方式是以虚拟内存页为单元,其优点在于可以借助于硬件与操作系统的支持来实现写屏障,从而不会给赋值器带来任何直接负担,但与卡表类似,回收器的工作负担则会加重。与卡表的不同之处在于,由于操作系统不可能获取对象的布局信息,因而页标记方案通常只能对应被修改的指针域,而无法获取其所属的对象。

第三,是否允许记忆集包含重复条目。允许重复条目的好处在于可以降低赋值器的去重检测开销,但代价是增大了记忆集的大小以及回收器处理重复条目的开销。卡表和页标记技术是通过在表中设置标记位或者标记字节的方式来进行标记的,因而其可以天然实现去重。如果使用记录对象的方式,也可以通过标记对象的方式实现去重,例如通过对象头部中的一个标记位来记录其是否已经添加到日志中,但如果以指针域为记录粒度则无法通过这一方式进行简单去重。尽管该策略可以降低记忆集的空间大小,但其需要赋值器执行一次额外的判断逻辑以及一次额外的写操作。如果不允许记忆集中出现重复对象,则记忆集的实现必须是真正的集合(set)而非多集合(multiset)。

综上所述,如果使用卡表或者基于页的记录策略,则回收器的扫描开销取决于脏卡或者脏页的数量。如果允许记忆集中出现重复条目,则回收器的开销将取决于指针写操作的数量,而如果不允许重复,则回收器的开销取决于被修改的指针域的数量。不论对于哪种情况,过滤掉回收无关指针都会减少回收器扫描根集合的开销。记忆集的实现方式包括哈希表、顺序存储缓冲区、卡表、虚拟内存机制与硬件支持,我们将逐一进行介绍。

193

11.8.3 哈希表

如果不希望在记忆集中出现重复条目,则其实现必须是真正的集合。同样,如果对象头部中没有足够的空间来记录其是否已经添加到记忆集,也需要通过集合来记录对象。我们进一步希望向记忆集中增加条目的操作可以很快完成,最好是在常数时间内。哈希表即是满足这些条件的实现方案之一。

在 Hosking 等 [1992] 的多分代内存管理工具包中,他们给出了一种基于线性散列环状哈希表(circular hash table)的记忆集实现方案,并将其应用在一种 Smalltalk 解释器中,该解释器将栈帧保存在堆的第 0 分代的第 0 阶中。具体而言,每个分代都会对应一个独立的记忆集,且记忆集中既可以记录对象,也可以记录域。其哈希表基于一个包含 $2^i + k$ 个元素的数组实现($k = 2$),它们将地址映射为一个 i 位的哈希值(从对象的中间几位中获取),并以此作为该地址在数组中的索引。如果该索引对应的位置为空,则将该对象的地址或域保存在该索引位置,否则将在后续的 k 个位置中查找可用位置(此时并非环状查找,正因如此,数组的大小才是 $2^i + k$)。如果依然查找失败,则对数组进行环状查找。

为减轻记录指针的工作量,写屏障首先过滤掉所有针对第 0 代对象的写操作以及所有新-新指针(即从新生对象指向新生对象的指针)的创建。另外,写屏障会将所有回收相关指针添加到一个单独的“草稿”记忆集中,而非直接将其添加到目标分代所对应的记忆集。该策略不会占用赋值器的时间来判断回收相关指针究竟属于那个记忆集,因而其可能更加适合多线程环境,除此之外,为每个处理器维护“草稿”记忆集也可以避免潜在的冲突问题,因为线程安全哈希表在运行时可能会引入较大开销。综上, Hosking 等使用 17 条内联 MIPS 指令来实现写屏障的快速路径,其中包括更新记忆集的相关调用。即使对于 MIPS 这种寄存器较多的架构,这一方案的开销也相对较高。在回收阶段,来自某一分代的根要么位于该分代对应的记忆集中,要么位于“草稿”记忆集中。回收器可以将分代所对应的记忆集中的回

收相关指针重新散列到“草稿”记忆集中，从而完成去重，然后再将“草稿”记忆集中的所有回收相关指针添加到合适的记忆集中。

Garthwaite 在其火车回收算法的实现中也使用了哈希表。其哈希表的操作一般是插入以及迭代，因而其使用开放定址法 (open addressing) 来解决冲突问题。由于哈希表中经常会记录相邻地址，所以其舍弃了会将相邻地址映射到哈希表中相邻槽的线性定址法 (即简单的地址模 N , N 为哈希表的大小)，取而代之的是通用的哈希函数。Garthwaite 选用了 58 位的质数 p ，并为每个哈希表绑定两个参数 a 和 b ，这两个参数是通过重复调用一个伪随机函数生成的 [Park and Miller, 1988]，且 $0 < a, b < p$ 。某一地址 r 在哈希表中对应的索引是 $((ar + b) \bmod p) \bmod N$ 。当冲突发生时，开放定址法需要一定的手段来进行再次探测。线性探测与平方探测 (下一个探测位置的索引值为当前索引值加 d ，且每次探测都对 d 增加一个常量 i) 可能会导致一组插入请求产生相同的探测序列，因而 Garthwaite 使用再散列方法，即把平方探测中的增量 i 替换为一个基于地址的函数。对于大小为 2 的整数次幂的哈希表，如果探测增量 i 为奇数，则可以确保整个哈希表都可以探测到。Garthwaite 的策略是：在每次探测时判断 d 是否为奇数，如果是，则将 i 设置为零 (线性探测)，否则则将 d 和 i 都设置为 $d + 1$ 。如此一来，可用探测序列的集合便可翻倍。最后，如果哈希表的负载过高，则需要扩展，一种可选方案是通过修改插入过程来重新平衡哈希表。当发生碰撞时，我们需要判断是要将当前正在插入的地址进行再次探测，还是将当前槽中原有的对象进行冲突探测 (并将其插入到新的位置)。Garthwaite 等人使用 robin hood 哈希 [Celis 等, 1985]，其每个槽中存储的条目都会记录其插入过程中的探测次数，由于哈希表所记录的地址中会存在很多为零的位 (例如卡的地址)，所以可以复用这些位来记录探测次数。当插入一个新地址时，如果探测所得的槽已被占用，我们会选择槽中的现有地址以及待插入地址中探测次数较多的一个留在槽中，而对另一个地址继续进行探测。

194

11.8.4 顺序存储缓冲区

使用简单的顺序存储缓冲区 (sequential store buffer, SSB) (例如内存块链表) 可以加快指针的记录速度。每个线程可以针对所有分代维护统一的本地顺序存储缓冲区，这样不仅可以避免写屏障选择适当缓冲区的开销，而且可以消除多线程之间的竞争。

一般情况下，向顺序存储缓冲区中添加条目只需要很少的指令：简单地判断 next 指针是否达到上限、将引用存入缓冲区中下一个位置、向前递增 next 指针。MMTk [Blackburn 等, 2004b] 使用内存块链表来实现顺序存储缓冲区，每个内存块的大小为 2 的整数次幂，同时也依照 2 的整数次幂对齐，其填充方向是从高地址到低地址。此时写屏障通过判定 next 指针的低位是否为零 (该操作通常很快)，便可简单地完成溢出检测。

有多种方法可以消除显式溢出检测的开销，如此一来，向顺序缓冲区中追加条目所需的指令可以降低到一至两条，如算法 11.4 所示。在 PowerPC 上，如果使用专用寄存器，则该操作可以通过一条指令完成：stwu fld, 4(next)。

算法 11.4 使用顺序存储缓冲区来记录指针

```

1 Write(src, i, ref):
2   add %src, %i %fld
3   st %ref, [%fld]                ; src[i] ← ref
4   st %fld, [%next]              ; SSB[next] ← fld
5   add %next, 4, %next           ; next ← next + 1

```


Appel [1989a], Hudson 和 Diwan [1990] 以及 Hosking 等 [1992] 使用写保护哨兵页 (guard page) 来消除显式溢出检测。当写屏障尝试在哨兵页上添加一个条目时, 陷阱处理函数会执行适当的溢出操作, 我们将在稍后详细讨论。触发以及处理页保护异常的开销很大, 其通常会花费数百甚至上千个指令, 因此只有当陷阱很少被触发时, 该策略才会体现出效率上的优势, 即陷阱执行开销应当小于 (大量) 软件检测所花费的开销:

$$\text{页保护陷阱的执行开销} \leq \text{溢出判断的开销} \times \text{缓冲区大小}$$

Appel 将顺序存储缓冲区保存在年轻分代中, 并使用链表来组织内存块, 据此可以确保在每个回收周期中页保护陷阱只会被精确地触发一次。Appel 将哨兵页布置在年轻分代末尾的保留空间中, 因此任何分配操作 (不论是分配对象还是记忆集内存块) 都可能触发陷阱并呼起垃圾回收。该技术要求年轻代的空间必须是连续的。某些系统可能会将堆布置在数据区的末尾, 并使用 brk 系统调用来扩大 (或者收缩) 堆。但正如 Reppy [1993] 所提到的, 为堆末端边界之外的页设置特殊的保护策略会干扰 malloc 函数对 brk 的调用, 因此更好的解决方案是使用更高的地址空间, 并使用 mmap 来管理堆的扩展与收缩。

195

某些体系架构所支持的特殊机制也可以用于消除溢出检测。例如 Solaris 系统的 UTRAP 异常, 该异常用于处理非对齐 (misaligned) 数据访问, 且速度要比 Unix 信号处理机制快上百倍。Detlefs 等 [2002a] 使用由 2^n 字节内存块组成的链表来实现顺序分配缓冲区, 每个内存块以 2^{n+1} 字节对齐但不满足 2^{n+2} 字节的对齐要求, 这可能造成一定的空间浪费。算法 11.5 描述了其插入流程: next 寄存器通常指向下一个条目之后 4 字节的位置, 当缓冲区被填满时 (即 next 寄存器指向 2^{n+2} 对齐边界之前的槽) 便会触发 UTRAP 陷阱, 正如示例中的第 5 行。

算法 11.5 基于非对齐数据访问实现边界检测

```

1 atomic insert(fld):
2   *(next - 4) ← fld           /* 在前一个槽中添加条目 */
3   tmp ← next >> (n-1)
4   tmp ← tmp & 6                /* tmp=4 或 6 */
5   next ← next + tmp

```

示例: $n = 4$ (4 字缓冲区):

```

insert at 32: next=40, next>>(n-1)=4, tmp=4
insert at 36: next=44, next>>(n-1)=5, tmp=4
insert at 40: next=48, next>>(n-1)=5, tmp=4
insert at 44: next=54, next>>(n-1)=6, tmp=6
insert at 50: UTRAP!

```

顺序存储缓冲区可以直接用于记忆集的实现, 也可以当作哈希表的快速记录前端。对于简单的两分代且使用集体提升策略的回收器, 次级回收完成后年轻代将被清空, 因而其可以简单地将记忆集抛弃, 因此在这一场景下, 回收器不需要更加复杂的记忆集结构 (假设顺序存储缓冲区在执行回收之前不会溢出)。但是, 其他更加复杂的回收器则需要在两次回收之间保留记忆集, 如果使用多分代, 即使定罪分代使用集体提升策略, 更高级别分代之间的分代间指针依然需要保留。如果定罪分代内部包含阶, 或者使用了其他延迟提升策略 (参见 9.4 节), 则记忆集依然需要保留从更老分代指向未提升对象的引用。

一种解决方案是简单地将顺序存储缓冲区中不再需要的条目移除, 可以将该位置的指针清空, 也可以将其指向只有在整堆回收时才会处理的对象 (或者永远不会回收的对象)。另外, 如果某一对象不包含任何回收相关指针, 则可以将其对应条目移除。但是, 这些解决方

案均无法控制记忆集的增长,且可能导致回收器不断重复处理相同的长寿条目。更好的解决方案是将需要保留的条目移动到各分代对应的记忆集中,这些目标记忆集可以使用顺序存储缓冲来实现,也可将其转换成更加精确的哈希表来记录。

11.8.5 溢出处理

哈希表以及顺序存储缓冲区均可能会溢出,有多种方案可以解决这一问题。当顺序存储缓冲区溢出时,MMTk 的解决方案是分配一个新的内存块并将其链接到顺序存储缓冲区中 [Blackburn 等, 2004b], Hosking 等 [1992] 的策略是无论是否会溢出,都将顺序存储缓冲区中的数据转移到哈希表中,并将前者清空。为保持哈希表相对稀疏,如果在插入一个指针时出现冲突,或者经历 k 次线性探测之后依然冲突,或者哈希表的使用率超过某一阈值 (例如 60%),则需将哈希表扩大。扩大的方式是增加键的长度并简单地将哈希表的大小翻倍,但如此一来键的长度便不能是编译常量,这将增加哈希表的大小以及写屏障的执行开销。Appel [1989a] 将其顺序存储缓冲区保存在堆中,一旦其发生溢出,则立即唤起垃圾回收,MMTk 也会在回收器自身元数据 (例如顺序存储缓冲区) 过大时发起回收。

11.8.6 卡表

卡表 (卡标记) 策略将堆在逻辑上划分为固定大小的连续区域,每个区域称之为卡 [Sobalvarro, 1988; Wilson and Moher, 1989a, b]。卡通常较小,介于 128 ~ 512 字节之间。卡表最简单的实现方案是使用字节数组,并以卡的编号作为索引。当某个卡内部发生指针写操作时,写屏障将该卡在卡表中对应的字节设置为脏 (如图 11.3 所示)。卡的索引号可以通过对指针域的地址进行移位获得。卡表的设计初衷在于尽量简化写屏障的实现并提高其性能,从而将其内联到赋值器代码中。另外,与哈希表或者顺序存储缓冲区不同,卡表不存在溢出问题。但这些收益总是要付出一定代价的:回收器的工作负荷会加重,因为回收器必须对脏卡中的域进行逐个扫描,并找出其中已被修改的、可能包含回收相关指针的域,此时回收器的工作量将正比于已标记卡的数量 (以及卡的大小),而非产生回收相关指针的写操作的发生次数。

使用卡表的目的在于尽可能减轻赋值器的负担,因而其通常应用在无条件写屏障中,这便意味着卡表必须能够将所有可能被 write 操作修改的地址映射到卡表中的某个槽。如果我们可确保堆中的某些区域永远不可能写入回收相关指针,同时引入条件检测来过滤掉这些区域的指针写操作,则可以减少卡表的大小。例如,如果将堆中高于某一固定虚拟地址边界的空间用作新生区 (回收器在每次回收过程中都处理该区域),则卡表只需要对低于该边界地址的空间创建对应的槽。

最紧凑的卡表实现方式应当是位数组,但多种因素决定了位数组并非最佳实现方案。现代处理器的指令集并不会针对单个位的写入设置单独的指令,因而位操作比原始操作需要更多的指令:读取一个字节、通过逻辑运算设置或清除一个位、写回该字节。更糟糕的是,这些操作序列并不是原子化的,多线程同时更新同一个卡表条目可能会导致某些信息丢失,即使它们所修改的是堆中不同的域或者对象。正因如此,卡表才通常使用字节数组。由于处理器清空内存的指令的执行速度更快,所以通常使用 0 来表示“脏”标记。在使用字节数组的场景下,在卡表中设置脏标记只需要两条 SPARC 指令 [Detlefs 等, 2002a] (其他架构所需的指令可能会稍多一些),如算法 11.6 所示。为方便表述,我们使用 ZERO 来代表 SPARC 寄存

器 %g0, 该寄存器的值通常为 0。BASE 寄存器的值需要初始化为 CT1-(H>>LOG_CARD_SIZE), 其中 CT1 为卡表的起始地址, H 为堆的起始地址, 两者均依照卡的大小 (即 512 字节) 对齐。Detlefs 等 [2002a] 使用一个 SPARC 本地寄存器来作为 BASE 寄存器, 并在程序进入到某一可能执行写操作的函数时设置其值, 该寄存器的值在函数调用时的保存则依赖寄存器窗口机制。

197

算法 11.6 SPARC 架构下基于卡表来记录指针

```
1 Write(src, i, ref):
2   add %src, %i, %fld
3   st %ref, [%fld]                                ; src[i] ← ref
4   srl %fld, LOG_CARD_SIZE, %idx      ; idx ← fld >> LOG_CARD_SIZE
5   stb ZERO, [%BASE+%idx]            ; CT[idx] ← DIRTY
```

Hölzle[1993] 进一步降低了大多数情况下写屏障的开销, 代价是牺牲了记录的精度, 如算法 11.7 所示。该算法中, 对卡表中第 i 个字节进行标记意味着从第 i 到第 $i + L$ 个卡都可能被修改过。如果某一对象中被修改的域在其内部的偏移量小于 L 个卡, 则可以在卡表中设置对象首地址所对应的字节。令 L 为 1 通常可以涵盖大多数指针写操作场景, 但数组则是一个例外, 写屏障必须采用传统的方式对其进行标记。如果使用 128 字节的卡, 则对于大小不超过 32 个字的对象, 对其内部任意一个域的修改均可以确保能将其首地址记录到卡表中。

算法 11.7 SPARC 架构下使用 Hölzle 的卡表策略来记录指针

```
1 Write(src, i, ref):
2   st %ref, [%src + %i]
3   srl %src, LOG_CARD_SIZE, %idx      /* 计算近似的字节索引 */
4   clrb [%BASE + %idx]                /* 清空字节图中的字节 */
```

只有当某个卡内部的最后一个对象所占用的空间会延伸到下一个卡时, 才可能发生歧义, 此时回收器可能还需要扫描该对象 (或者该对象必要的起始部分)。

即使卡的大小比较小, 卡表所占用的空间通常也可以接受。例如在 32 位体系架构下, 128 字节的卡所对应的卡表仅会占用堆中不到 1% 的空间。在确定卡的大小时需要在空间开销与回收器扫描根的时间开销两方面做出权衡: 增大卡的大小, 尽管会减少卡表的空间开销, 但其精度也会降低, 反之, 相反。

在回收阶段, 回收器必须在所有脏卡中查找回收相关指针, 因而其必须先对卡表进行扫描并找出脏卡。由于赋值器的更新操作通常具有较高的局部性, 所以干净的卡与脏卡通常会出现聚集效应, 回收器可以根据这一特性来加速查找过程。如果卡表使用字节数组来实现, 则回收器可以一次性对卡表中由 4 个或者 8 个槽所组成的字进行检测。

如果分代回收器不使用集体提升的策略, 则在次级回收之后, 某些年轻代存活对象会留在年轻代里, 其他的则会得到提升。如果得到提升的对象引用了尚未提升的对象, 则由此产生的老 - 新指针不可避免地会将某个卡打上脏标记。但是这些被已提升对象所引用的未提升对象终究都会得到提升, 因此我们应当尽可能不去标记已提升的对象所在的卡, 否则在下一轮回收中将会出现一些不必要的卡扫描。在将对象提升到某一干净的卡时, Hosking 等 [1992] 使用过滤复制屏障来扫描得到提升的对象, 因此其仅会在必要时才将卡标记为脏。

198

尽管如此, 如果堆空间过大, 回收器依然可能需要花费大量时间来跳过干净的卡。

Detlefs 等 [2002a] 观察发现, 绝大多数卡都是干净的, 且单个卡中很少会包含超过 16 个分代间指针。因此可以使用两级卡表来加速回收器查找脏卡的过程, 尽管这一策略会付出额外的空间开销。第二级卡占用的空间更小, 其中的每个槽对应 2^n 个粒度更细的卡, 因而其能够将干净卡的扫描速度提升 n 倍。写屏障可以使用与算法 11.6 类似的技术实现 (只需要额外增加两条指令), 但其需要确保第二级卡表的起点与第一级对齐, 即 $CT1 - (H \gg LOG_CARD_SIZE) = CT2 - (H \gg LOG_SUPERCARD_SIZE)^\ominus$, 如算法 11.8 所示。这一要求可能会造成一定的空间浪费。

算法 11.8 SPARC 架构中的两级卡表

```

1 Write(src, i, ref):
2     add %src, %i, %fld
3     st %ref, [%fld]                                /* 执行写操作 */
4     srl %fld, LOG_CARD_SIZE, %idx                 /* 获取一级索引 */
5     stb ZERO, [%BASE+%idx]                        /* 将一级卡标记为脏 */
6     srl %fld, LOG_SUPERCARD_SIZE, %idx            /* 获取二级索引 */
7     stb ZERO, [%BASE+%idx]                        /* 将二级卡标记为脏 */

```

11.8.7 跨越映射

在回收阶段, 回收器必须对其在卡表中找到的脏卡进行处理, 这一过程需要确定卡中被修改的对象及对象内部被修改的槽。扫描对象中的域通常只能从对象的起始地址开始, 但卡的起始地址却不一定与对象的起始地址重合, 因而扫描过程并不能直接进行。更加糟糕的是, 导致卡被标记为脏的指针域可能属于某一大对象, 而该对象的头部则可能位于该卡之前的某个卡中 (这也是需要对大对象进行分离存储的原因之一)。为确保回收器可以从对象头部开始扫描, 我们必须借助于跨越映射来描述对象在卡内部或者卡之间的布局。

跨越映射中的每个条目与卡表中的卡是一一对应的关系, 其每个条目所记录的是对应卡中第一个起始地址落入该卡的对象在卡中的偏移量。回收器会在提升对象时设置年老代卡所对应的跨越映射条目, 如果分配器直接将对象预分配在年老代则也需设置这一信息。新生区的对象不可能指向更年轻的对象 (它们已经是最年轻的对象), 因而无需为其维护卡表。卡表的记录方式 (记录对象或是记录被修改的域) 决定了跨越映射的设计方式。

如果写屏障使用卡表来记录被修改的指针域, 则跨越映射必须记录每个卡中最后一个 (起始地址落入该卡的) 对象的偏移量, 如果任何一个对象的起始地址都不在该卡中, 则跨越映射必须记录一个负数偏移量。由于对象可能会跨越多个卡, 所以被修改槽所属对象的起始地址可能会位于脏卡之前的另一个卡中。例如在图 11.3 中, 堆中的白色方框表示对象, 假设图中所描述的场景是在 32 位环境下, 且卡的大小为 512 字节。第一个卡中最后一个对象的偏移量为 408 字节 (102 个字), 该值将会记录到其在跨越映射的对应的条目中。该对象跨越了 4 个卡, 因而跨越映射中后面两个条目的值均为负数。当堆中第 5 个对象的某个域被修改之后 (灰色所示区域), 其所对应的卡 (第 4 个卡) 将被标记为脏 (黑色区域)。为找到被修改对象的起始地址, 回收器必须从跨越映射中进行后退查找, 直到发现某一偏移量非负的条目为止 (见算法 11.9)。需要注意的是, 负数表示需要后退的距离, 当对象较大时, 回收器可以根据该值快速找到对象首地址所在的条目。当然, 系统也可以在这些条目中填入特定的值来表示倒退, 例如 -1, 但这将减缓回收器在大对象中的查找速度。

[199]

\ominus 这两个值都需要占用独立的寄存器。——译者注

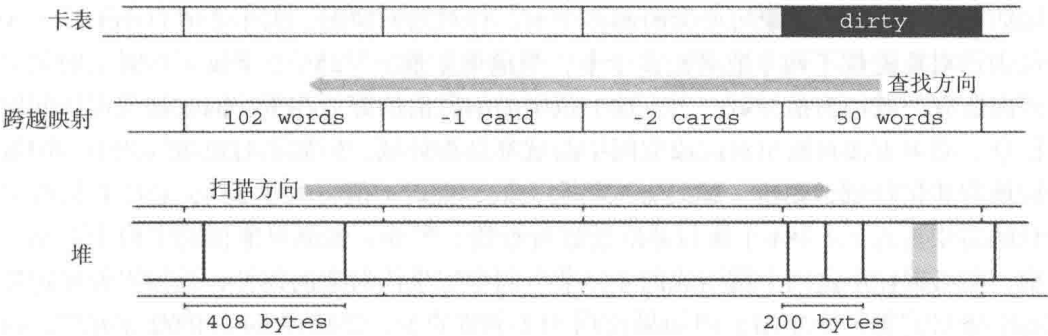


图 11.3 记录被修改指针域的卡表及其所对应的跨越映射。图中有一个卡已经标记为脏（黑色的卡），被修改的指针域为灰色，跨越映射记录了卡中最后一个（起始地址落入该卡的）对象的偏移量（以字为单位）

年老代通常使用非移动式回收算法进行管理，此时空闲内存块与已分配内存块便会在堆中混杂分布。在并行回收器中，为避免回收过程中可能存在的冲突，不同的回收线程往往会拥有不同的目标提升区域，因此得到提升的对象很容易形成多个孤岛，每个孤岛之间都是较大的空闲区域。为了更好地支持堆的可解析性，每个空闲区域可以使用一个自描述的伪对象来填充。但是，基于槽的跨越映射算法却更适用于堆中对象排布比较密集的情况：如果在两个脏卡之间存在一块很大的空闲内存块（例如 10MB），则算法 11.9 中 `search` 方法的第一个循环可能需要迭代数万次，才能找到用于描述空闲内存块的伪对象的头部。降低这一查找开销的方法之一是在跨越映射中存储后退距离的对数值，即如果某个条目所记录的值为 $-k$ ，则意味着回收器需要后退 2^{k-1} 个卡，然后根据新位置中所记录的值继续执行查找（与线性后退策略相似）。如果需要在 大块空闲内存的起始地址分配对象，回收器只需要更新 $\log(n)$ 个跨越映射条目，即可修正跨越映射的状态，其中 n 为此内存块所占据的卡的数量。

算法 11.9 基于跨越映射来查找被修改槽所属的对象（`trace` 为回收器的标记或者赋值子过程）

```
1 search(card):
2   start ← H + (card << LOG_CARD_SIZE)
3   end ← start + CARD_SIZE                                /* 下一个卡的起始地址 */
4   offset ← crossingMap[card]
5   while offset < 0
6     card ← card + offset                                  /* 偏移量为负数：继续后退 */
7     offset ← crossingMap[card]
8   offset ← CARD_SIZE - (offset << LOG_BYTES_IN_WORD)
9   next ← H + (card << LOG_CARD_SIZE) + offset
10  repeat
11    trace(next, start, end)                               /* 对位于 next 处的对象进行追踪 */
12    next ← nextObject(next)
13  until next ≥ end
```

Garthwaite 等 [2006] 设计出一种巧妙的跨越映射编码方式，该策略可以消除查找过程中的循环。该策略中，我们可以简单地将跨越映射中的每个条目 v 看作是 16 位无符号整数（两个字节）。表 11.3 描述了其编码策略。如果 v 的值为零，意味着其所对应的卡中任何对象都不包含引用。如果 v 的值不大于 128，则该值表示卡中第一个对象与卡的末端之间的距离（单位为字）。需要注意的是，此处的记录方式与图 11.3 中所描述的记录方式有所不同，记录第一个对象而非最后一个对象的偏移量可以确保回收器在大多数情况下无须后退到前一个

卡。诸如数组之类的大对象可能会跨越多个卡，针对这一情况，大于 256 且小于等于 384 的编码值表示对象跨越了两个或者更多个卡，当前卡中前 $v - 256$ 个字属于该对象的末尾，且此空间内所有的域均为指针域。引入这一范围的编码值的好处在于，回收器无需访问对象的类型信息，便可直接判断出对此段空间中的域都是指针域。但如果对象落入该卡中的域混杂着指针域和非指针域，则这一编码方式将会失效，此时 v 值将大于 384，意味着回收器应当在跨越映射中后退 $v - 384$ 个条目并继续进行查找。另外，如果对象横跨了两个完整的跨越映射槽，则可以在由这两个槽组成的 4 字节空间中记录该对象的地址，该方案假定跨越映射中的每个条目占据两个字节，但如果使用 512 字节的卡，并使用 64 位的对齐方式，则仅用一个字节也可达到同样的编码效果。

表 11.3 Garthwaite 等人的跨越映射编码方式

值 (v)	编码含义
$v = 0$	对应的卡中不包含指针
$0 < v \leq 128$	卡中第一个对象距卡的末端的偏移量为 v 个字
$256 < v \leq 384$	卡中前 $v - 256$ 个字属于某一对象的末尾，且此空间内所有的域均为指针域
$v > 384$	后退 $v - 384$ 个卡并继续进行查找

11.8.8 汇总卡

某些分代回收器并不采用集体提升策略，因此如果回收器通过对脏卡的扫描发现了回收相关指针但并未将其目标对象提升，则回收器需要保留该卡的脏标记，以便后续过程再次进行扫描。如果后续回收过程可以直接获取此类脏卡中的回收相关指针而不用再对卡表进行扫描，则可以提升回收效率。幸运的是，绝大多数脏卡中都只包含数量很少的回收相关指针，因此 Hosking 和 Hudson[1993] 建议在完成某个卡的扫描之后将其中的回收相关指针添加到哈希表中，同时清除该卡的脏标记。Hosking 等 [1992] 也采用相同的策略，不同之处在于其使用的是顺序存储缓冲区。

Sun 的 Java 虚拟机中，清扫器会对清扫完成后依然包含回收相关指针的卡进行汇总 (summarise)，并据此优化卡的再扫描过程 [Detlefs 等, 2002a]。由于卡表的实现方式是字节数组而非位数组，所以可以将卡的状态进一步划分为“干净”、“已修改”、“已汇总”。如果回收器在“已修改”的卡中发现不多于 k 个回收相关指针，则将该卡标记为“已汇总”，并将这些指针域的偏移量记录在“汇总表”(summary table) 的对应条目中。如果卡中回收相关指针的数量大于 k (例如 $k = 2$)，则该卡将依然保持“已修改”状态，同时其在汇总表中对应的条目将被标记为“已溢出”。因此在下次回收过程中，回收器无需用跨映射对卡进行扫描便可直接找到其中的回收相关指针 (除非该卡重新被写屏障标记为脏)。另外，由于卡表的实现方式是字节数组，所以如果卡相对较小，也可直接在卡表中记录少量的偏移量信息。

Reppy[1993] 在卡表的编码中加入额外的分代信息以降低扫描开销。当完成某个卡的清理后，其多分代回收器会获取该卡内部所有指针域引用的对象所处的分代，并将其中最年轻分代的编号 (0 代表新生代) 记入汇总卡。因此在后续对第 n 个分代的回收过程中，如果某个卡在汇总卡中的对应条目的值大于 n ，则回收器可以快速将其跳过。对于使用 5 个分代、卡大小为 256 字节的 Standard ML 堆，该策略可以节约 80% 的回收时间。

[201]

11.8.9 硬件与虚拟内存技术

某些早期的分代垃圾回收器需要依赖操作系统以及硬件的支持。支持带标签值 (tagged

value) 的硬件架构可以轻易区分出指针与非指针, 某些硬件写屏障还可以在页表中进行置位操作 [Moon, 1984]。在没有特殊硬件支持的条件下, 也可以借助于操作系统来实现对写操作的追踪。例如 Shaw[1988] 对 HP-UX 操作系统进行修改, 并利用其换页系统来达到这一目的。虚拟内存管理器通常需要对脏页进行记录, 并以此判定在某一页被换出时是否需要将其写回到交换文件 (swap file)。Shaw 的修改会拦截虚拟内存管理器的换页操作, 并记录被换出的页的脏标记状态, 他同时添加了数个系统调用来清空一组页的脏标记, 或者返回自从上次回收以来被修改的页集合。该策略的优势在于其不会给赋值器引入任何常规开销, 但其缺点也十分明显: 操作系统将某一页标记为脏时不可能区分写入的值是否为指针, 因此其记忆集的精度较低, 同时换页陷阱与系统调用的开销也不容忽视。

为避免对操作系统进行修改, Boehm 等 [1991] 在一轮回收之后会修改已回收内存页的写保护策略。在该页发生的第一个写操作会触发写保护异常, 陷阱处理函数会设置该页的脏标记, 并解除该页的写保护策略, 以避免在下一轮回收之前在该页重新触发陷阱。在回收过程中, 回收器显然需要解除对象将被提升到的目标页的写保护策略以避免触发陷阱。页保护策略不会给赋值器带来开销, 且与卡表类似, 写屏障的开销将正比于被修改的页的数量, 而与写操作的数量无关。但是, 该策略却引入了其他更加昂贵的开销: 从操作系统读取脏页信息的开销通常较大; 页保护机制有可能引发所谓的“陷阱风暴” (trap storms) 问题, 即在回收完成之后赋值器会触发大量写保护异常来解除程序工作集的写保护 [Kerny and Petrank, 2006]; 页保护异常本身的开销就不容忽视, 如果其处理函数在用户空间执行则开销更大; 操作系统页通常会比卡大得多, 因而页扫描算法需要更加高效 (或许可以使用类似于汇总卡的技术来提升扫描性能)。

11.8.10 写屏障相关技术小结

Hosking 等 [1992] 以及 Fitzgerald 和 Tarditi[2000] 各自发现, 对于分代回收器而言, 没有哪种记忆集实现机制可以声称比其他机制更优 (但他们都没有考虑 Sun 的汇总卡技术或者类似技术)。基于页保护策略的记忆集性能最差, 但如果缺乏编译器的支持, 这可能是对写操作进行追踪的唯一方式。对于卡表记忆集而言, 卡的大小为 512 字节时通常性能最佳。

Blackburn 和 Hosking [2004] 统计了不同的分代间写屏障在不同平台上的执行开销。他们对卡标记以及 4 种局部屏障 (partial barrier) 机制进行了研究, 即边界检测 (boundary test)、日志检测 (logging test)、帧比较 (frame comparison)、混合屏障 (hybrid barrier)。为了比较不同局部屏障的性能, 他们排除了在记忆集中插入元素的开销。边界检测会判定指针是否跨越了某一在编译期就已经确定的空间边界; 日志检测会判断对象头部中的“日志”域; 帧屏障能判断某一指针是否跨越了两个大小为 2^n 且以 2^n 对齐的空间, 方法是对指针的来源地址与目的地址执行异或操作。这些屏障技术均可提升回收器在选择定罪空间时的柔性 [Hudson 和 Moss, 1992; Blackburn 等, 2002]。最后, 混合屏障使用静态边界检测来处理数组, 而用日志检测来处理纯对象。

他们得出的结论是: 写屏障的开销 (不包括在使用局部屏障技术时向记忆集插入元素的开销) 通常很小, 一般不会超过程序整体执行时间的 2%。即使对于开销稍大的写屏障, 其引入的开销也很容易被分代垃圾回收器所带来的整体性能提升所弥补 [Blackburn 等, 2004a]。但是, 写屏障在不同的平台上 (Intel Pentium 4, AMD Athlon XP 以及 PowerPC 970) 也会存在性能的差异, 特别是对于帧屏障和卡表。例如, 帧屏障在 x86 平台上的开销

会明显比其他平台要大，而在 PowerPC 平台上的开销最低；Blackburn 和 Hosking 发现，在 x86 平台上进行异或操作需要使用 `eax` 寄存器，这可能会增大寄存器的压力。另外，卡标记在 PowerPC 平台上的开销会比局部屏障技术昂贵的多（编译器生成的指令序列会比本书中介绍的要长得多）。最后，我们建议在选择写屏障时一定要使用真实的基准测试程序，并在多个硬件平台上仔细进行实验，不同平台上的最佳实现策略可能会有所不同。

11.8.11 内存块链表

数组的优势在于其不需要为每个元素维护链接表指针或者对象头部，且具有较好的局部性，但数组也会面临大量空间都被浪费的情况。另外，移动数组或者增大数组的开销通常很大。因此，类似链表的数据结构在垃圾回收器中应用十分普遍，分代式回收器中的记忆集即是这样一个例子。内存块链表（chunked list）可以将数组和链表的优势相结合，它既保证了存储的密度，也无需进行重新分配，空间浪费率以及运行时开销也相对较小。该数据结构是由内存块彼此链接而构成的，它们通常以双向链表的形式链接成双向队列。每个内存块是由用于存储数据的槽数组外加一到两个链接指针组成的，如图 11.4 所示。

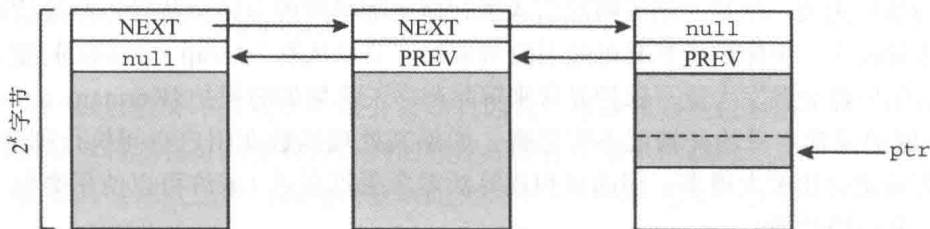


图 11.4 以内存块链表方式实现的栈。阴影区的每个槽都包含数据，每个内存块以 2^k 字节为边界进行对齐

该数据结构还有一种十分优雅的组织形式：我们可以令内存块的大小为 2 的整数次幂，即 2^k ，同时也依照 2^k 地址边界进行对齐。如此一来，我们仅需要一个指针便可实现内存块链表的扫描、插入、移除（传统的实现方案需要一个“当前内存块”指针外加当前内存块内部的迭代索引号）。算法 11.10 展示了在内存块链表中进行双向遍历的代码，其所使用的便是这一技术，其中的取模运算可以通过移位和掩码操作来实现。

算法 11.10 内存块链表的遍历

```

1  /* 假设每个内存块的大小为  $2^k$ ，同时也依照  $2^k$  地址边界进行对齐 */
2  /* 假设指针与槽的大小均为 4 字节 */
3  NEXT = 0 /* 指向下一个内存块中数据起始地址的指针在内存块中的字节偏移量 */
4  PREV = 4 /* 指向上一个内存块中数据末端地址的指针在内存块中的字节偏移量 */
5  DATA = 8 /* 数据起始地址在内存块中的字节偏移量 */
6
7  bumpToNext(ptr):
8      ptr ← ptr + 4
9      if (ptr %  $2^k$ ) = 0 /* 到达本内存块的末端 ... */
10         ptr ← *(ptr -  $2^k$  + NEXT) /* ... 将指针移动到下一个内存块数据区的首地址 */
11         return ptr
12
13  bumpToPrev(ptr):
14      ptr ← ptr - 4
15      if (ptr %  $2^k$ ) < DATA /* 已越过本内存块中数据区的首地址 ... */
16         ptr ← *ptr /* ... 将指针移动到上一个内存块数据区的末端地址 */
17         return ptr

```

内存块链表的一个重要的使用场景与并行回收有关。如果使用内存块链表来实现工作队列，则每个回收线程便能够以内存块而非单个元素为单位来获取工作。如果内存块足够大，则不同回收线程从工作队列中获取工作时的冲突便可大大缓解。反过来看，如果内存块足够小，则很容易实现各回收线程之间的负载均衡。内存块链表的另一个使用场景是本地分配缓冲区（参见第 7.7 节），此时链表中的每个内存块均为空闲内存。

11.9 地址空间管理

在前面的章节中，我们介绍过多种算法以及堆布局方式，其中的某些会影响系统对虚拟地址空间的使用。某些算法要求使用大块连续地址空间，或者在这一场景下实现更为简单。在 32 位地址空间下，如果使用静态布局方式，系统通常很难保证各个空间的大小能够满足所有应用程序的要求。更加糟糕的是，操作系统有可能将动态链接库（也称共享对象文件）加载到地址空间的任意位置，造成空间的割裂，从而进一步增大了大块连续地址空间的获取难度。另外，出于安全目的，操作系统可能会将动态库加载在地址空间的随机位置，因此程序每次运行时，动态库的位置便会有所不同。64 位的大地址空间是这一问题的解决方案之一，但更大的指针同时也会增大应用程序所占用的物理内存。

203

使用大块连续地址空间布局策略的主要原因之一是确保基于地址比较的写屏障的执行效率，即写屏障可以将指针与一个固定地址或者另一个指针直接进行比较，而无需进行额外的查表操作。例如，如果将分代系统中的新生区布置在堆空间的一端，则写屏障只需一次简单的地址比较，便可判断出写入堆的指针是否引用了位于新生区的对象。

在设计一个新系统时，应当尽量避免把堆设计成大块连续地址空间的形式，而应当设计成基于帧（frame）的形式，或者至少允许在连续地址空间中存在“空洞”。但不幸的是，这一要求可能会导致写屏障不得不借助于查表操作。

假设查表操作的开销可以接受，则系统可以将逻辑地址空间映射到可用虚拟地址空间，从而能够管理更大的逻辑地址空间。尽管该策略并不能增加堆空间的大小，但其确实可以避免系统对地址空间连续性的依赖，因此给系统的设计提供了一定的柔性。该策略将可用内存划分成大小为 2 的整数次幂且依照 2 的整数次幂对齐的帧，每一帧的大小通常会大于一个虚拟内存页。系统使用一张表来维护所有的帧，并以帧的编号（通常是帧首地址的高位）作为索引以记录其逻辑地址，各种面向地址的写屏障便可基于该表进行地址比较。对于分代间写屏障，系统还可以将每个帧所处分代的编号记录到表中。算法 11.11 给出了此类写屏障的伪代码，代码中的每一行在一般的处理器上都对应一条指令，如果帧表中的每个条目都对应一个字节，则可以简化数组的索引操作。需要注意的是，即使 ref 为空，该算法也能正常工作，为达到这一目的，我们可以简单地为地址为零的帧赋予最高的分代编号，由此代码在执行过程中便可避免对地址为零的帧调用 remember 方法。

204

算法 11.11 基于帧的分代间写屏障

```

1 Write(src, i, ref):
2   ct ← frameTableBase
3   srcFrame ← src >>> LOG_FRAME_SIZE
4   refFrame ← ref >>> LOG_FRAME_SIZE
5   srcGen ← ct[srcFrame]
6   refGen ← ct[refFrame]
7   if srcGen > refGen
8     remember(src, &src[i], ref)
9   src[i] ← ref

```

我们甚至可以将较大地址空间内的多块可用内存“合并”成较小的连续空间——操作系统正是以这种方式来为进程提供虚拟内存的。一种实现策略是使用宽地址并检查每个地址空间访问操作，这相当于是使用软件来模拟虚拟内存管理硬件的工作，其中可能会包括软件层面实现的转译后备缓冲区等。该方案的性能惩罚可能相当高，当然也可以通过对虚拟内存硬件施加影响来避免惩罚，第 11.10 节将介绍这方面的更多细节。

在构建系统时最好能确保堆在系统启动时便具备迁移能力。许多系统都存在一个起始堆 (starting heap) 或者系统映像 (system image)，系统在启动时便会加载它们。该映像通常假定其自身会常驻在某一特定的地址空间中，但如果该地址已被动态链接库所占据，则其加载过程便会遇到问题。因此，如果系统映像内部用一张表来记录当自身被移动时哪些字需要做出调整（其实现方案与许多代码段的加载十分类似），映像加载器便可相对直接地将其移动到地址空间中的其他位置。同理，如果整个堆或者部分堆空间具有迁移能力，也能提升系统的柔性。

在实际应用中，在进行虚拟内存管理时，我们可以仅为托管系统保留特定的地址空间，但并不要求操作系统为其分配真正的内存页，从而可以避免操作系统在运行时将动态链接库映射到保留地址空间。这些页通常都是请求二进制的零页。这一操作的开销相对较低，但可能会影响操作系统的资源保留（例如交换空间），并且所有虚拟内存映射操作的开销通常都比较大。当堆空间较大时，程序也可以通过提前分配页来判定系统中的剩余资源是否足够，但操作系统在请求二进制零页真正被访问之前通常不会为其分配资源，因此简单地进行页分配可能会得出错误的预判。

11.10 虚拟内存页保护策略的应用

垃圾回收系统可以借助于虚拟内存页保护检查机制实现多种检测，这一检测方式在常规情况下的开销很低甚至没有开销，同时也不需要检测过程中增加显式的条件分支。但是，使用该方案时必须考虑陷入页保护陷阱的开销，陷阱处理函数需要先陷入操作系统再返回用户态，因此其开销可能相当大。另外，修改页保护策略也会具有一定开销，特别是在多处理器环境下，因为系统可能需要挂起所有正在运行的处理器并更新它们的内存页映射信息。因此在某些情况下，即使可以使用页保护陷阱，使用显式判断的开销也通常更低 [Hosking 等，1992]。陷阱在处理“不合作”的代码时依然有用，因为除此之外，系统无法通过其他方法来实现屏障或者检测。

另外一种需要考虑的情况是，出于硬件性能方面的原因，内存页的大小在未来可能会进一步增大，同时开发者所使用的内存总量也越来越多，系统的可映射内存也越来越多。但是，基于速度和能耗方面的考虑，转译后备缓冲区的大小却不太可能进一步增大。由于转译后备缓冲区的大小或多或少都是固定的，因此如果使用较小的页，转译后备缓冲区查找不命中的概率就会增大，但如果使用更大的页，某些虚拟内存相关技巧可能不再适用。

我们假设在某一体系架构中，页保护策略包括读写访问、只读访问、禁止访问这三种。此处我们不关注可执行权限，因为我们尚未发现在垃圾回收技术中使用不可执行保护的案例，另外，某些平台可能不支持对可执行权限的控制。

11.10.1 二次映射

在介绍页保护策略的具体应用之前，我们先描述二次映射 (double mapping) 技术，系

统可以通过该技术将相同的页以不同的保护策略映射到不同的虚拟地址。我们以遵守目标空间不变式的增量复制回收器为例（参见第 17 章）。为阻止赋值器访问尚未完成处理的内存页中的来源空间指针，回收器可以将这些页的保护策略设置为禁止访问，相当于是借助于硬件支持高效地创建了一个读屏障。但如此一来，回收器如何才能处理这些页？在并发系统中，如果回收器解除这些内存页的禁止访问保护，则赋值器可能会在回收器处理完成之前访问到页中的内容。为解决这一问题，回收器可以将待处理页以读写访问权限二次映射到其他地址，此时回收器便可基于其第二个映射来进行内容处理，处理完成后，回收器便可解除该页的禁止访问保护，并恢复所有等待访问该页的赋值器线程的执行。

当地址空间较小时（即便是 32 位在当前也可以算作小地址空间），进行二次映射可能存在困难。一种解决方案是 fork 出一个子进程进行处理，子进程将以不同的页保护策略来对待处理页映射到自身的地址空间，回收器可以通过与父子进程之间的通信来引导子进程完成页的处理。

需要注意的是，二次映射技术在某些系统中可能遇到问题。如果高速缓存依照虚拟地址进行索引，则可能出现潜在的高速缓存不一致问题，因为此时进行二次映射的地址很可能出现高速缓存不一致问题。为避免这一问题，硬件系统通常不允许别名条目（aliased entries）同时驻留在高速缓存中，但这可能导致额外的高速缓存不命中问题。不过在我们的应用场景中，赋值器和回收器一般是在相邻时间访问同一内存页的两个映射，且运行在同一个处理器上（因此高速缓存不命中问题就不那么重要——译者注）。另外，如果系统使用倒排页表，则每个物理内存页在任意时刻只能映射到一个虚拟地址上，此时系统便无法支持二次映射。这种情况下，操作系统可以快速将某一物理内存页的虚拟地址失效并将其与另一个虚拟地址关联，但这可能引发高速缓存刷新操作。

11.10.2 禁止访问页的应用

在对二次映射的描述中，我们已经看到了禁止访问保护策略的一个应用，即无条件读屏障。该策略至少还有两种常见的应用场景。一是探测程序对空指针（即目标地址为 0 的指针）的解引用操作，即系统将第 0 页（以及其后的几个页）设置为不可访问页，如果赋值器尝试访问空指针所指向的域，则其必然会对不可访问的页执行读或者写操作。由于对空指针解引用异常的处理通常不需要很快，所以在这一场景下使用禁止访问页保护策略较为合理。极少数情况下程序会访问距 0 地址偏移量较大的地址，编译器可以针对这一情况增加显式检测。如果将对象的头部或者其他域布置在对象指针地址负数偏移量的位置，则系统也可对地址最高的几个页做禁止访问保护（0 地址的负数偏移量将绕回到最高地址——译者注）。但在大多数系统中，高地址空间通常会保留给操作系统使用。

禁止访问页保护策略的另一个常见的应用场景是哨兵页（guard page）。例如，以顺序存储缓冲区作为实现的记忆集在插入新元素时需要经历三个步骤：判断缓冲区的剩余空间是否足够、将新元素写入缓冲区、增加缓冲区指针。如果在缓冲区的末端布置一个禁止访问的哨兵页，则写屏障可以省去检测剩余空间以及调用缓冲区溢出处理子过程的操作。由于写屏障的调用频率通常较高，且其代码可能会被嵌入在很多位置，因而哨兵页技术可以加速赋值器的执行速度并减小代码体积。

某些系统使用相同的策略来检测栈或堆的溢出，即在栈（堆）的末尾布置一个哨兵页。检测栈溢出的最佳方式是在子过程开始执行时立即尝试访问其将建立的新栈帧的最远处。此

时一旦触发哨兵页陷阱, 指令指针将位于一个事先预定的位置, 因而陷阱处理函数可以通过重新分配的方式增大栈空间, 或者增加一个新的栈分段并调整栈帧指针, 然后再恢复赋值器的执行。类似地, 当使用顺序分配缓冲区时, 分配器可以在执行分配之前访问新对象的最后一个字, 一旦该字落入缓冲区末尾的哨兵页中, 分配器将触发一个陷阱。

不论在哪种情况下, 如果新的栈帧或者对象过大以至于最远的一个字可能会越过哨兵页, 则系统仍需使用显式的边界检查。但如此巨大的栈帧和对象在许多系统中都十分罕见, 且大对象通常会花费更多的时间初始化使用, 这一开销通常足以掩盖显式边界检查的开销。

我们还可以利用禁止访问页保护策略在较小虚拟地址空间中获取较大的逻辑地址空间, Texas 持久对象存储 [Singhal 等, 1992] 即是一个案例。尽管该策略是针对数据持久化而设计的 (程序下次执行时, 堆中的数据依然保持着上一次运行结束时的状态), 但其所用到的技术也同样适用于垃圾回收等非持久化场景。该系统基于内存页进行工作, 每个页的大小与虚拟内存页相同, 或者是后者的 2ⁿ 倍。系统通过一张表来记录每个逻辑页的状态, 每个页不仅有其在 (虚拟) 内存中的地址, 系统还会为其在磁盘上维护一个明确的托管交换文件。每一页都可以有以下四种状态:

- 未分配 (unallocated): 页为空, 尚未得到使用。
- 驻留 (resident): 页中的数据已经加载到内存, 并且可以访问; 但其在磁盘上对应的交换文件不一定存在。
- 非驻留 (non-resident): 页中的数据在磁盘上, 无法直接访问。
- 保留 (reserved): 页中的数据在硬盘上, 无法直接访问, 但已为其保留虚拟地址。

新创建的页的初始状态为“驻留”, 且系统会为其分配新的逻辑地址 (与虚拟内存地址无关)。随着虚拟内存的不断使用, 某些页可能需要换出到磁盘。保存过程需要基于页的逻辑地址, 系统需要将页中所有的指针转换成更长的逻辑地址, 因而其在硬盘中的存在形式一般会比其在内存中的要大。这一过程在文献中被称为逆转换 (unswizzling) [Moss, 1992], 它要求系统必须能够准确地找出每个页中的指针。在“驻留”页被换出后, 其状态将变成“保留”, 系统进一步将其对应的虚拟地址空间设置为禁止访问, 此时一旦程序访问被换出的页便会触发页保护陷阱, 陷阱处理函数会将该页重新载入内存。

如果系统需要复用“保留”页的虚拟地址空间, 则其必须确保该“保留”页不被任何“驻留”页引用。为达到这一目的, 系统可以将所有引用该页的“保留”页换出, 然后将该页的状态修改为“非驻留”, 此时系统便可复用其地址空间。

需要注意的是, “驻留”页只能引用“驻留”页或者“保留”页, 但不能直接引用“非驻留”页中的数据。

接下来我们考虑程序在访问“保留”页时的情形 (如果某一可达对象位于被换出的页中, 则该页的状态必然为“保留”)。系统通过查表获取该页的逻辑地址, 并将其从磁盘加载到内存。然后系统需要遍历其中的逻辑地址, 并将其转换成较短的虚拟地址 (该过程称为指针转换 (pointer swizzling))。对于该页中指向“驻留”页或者“保留”页中的引用, 转换操作均可直接通过查表完成; 但是, 对于其中指向“非驻留”页的引用, 系统必须先为目标页保留虚拟地址 (此时目标页的状态将从“非驻留”转变为“保留”), 然后再将这些引用从逻辑地址转换为虚拟地址。为这些新的“保留”页分配虚拟地址可能需要将其他页换出, 该操作可能进一步将被换出页的状态修改为“非驻留”并回收其虚拟地址空间。

与操作系统的虚拟内存管理器一样, Texas 也需要一种高效的换页策略, 因此其可以理

所当然地借鉴虚拟内存管理的相关算法。

如何在垃圾回收环境下使用这一策略?很显然,对一个比虚拟地址空间还大的堆进行整堆回收的开销极大。关于如何对持久存储进行回收这一问题,目前存在这方面的文献,本书不打算详细展开。但可以肯定是,基于分区的回收策略在该场景下十分有用,同时类似于成熟对象空间的技术(Mature Object Space) [Hudson and Moss, 1992] 可以确保回收的完整性。

与 Texas 相关的回收技术包括书签回收器(Bookmarking collector) [Hertz 等, 2005; Bond 和 McKinley, 2008], 但书签回收器的主要目的在于避免回收过程给物理内存造成颠簸,它并未在虚拟地址之外引入逻辑地址。书签回收器会对所有被操作系统换出的页中的引用进行汇总,因此回收器便可避免访问已经换出的页并维持现有的工作集合。与分代回收器中的记忆集类似,回收工作的精确度可能有所下降,因为回收器可能会(依照汇总信息)对已换出页中死亡对象内的指针进行遍历。

11.11 堆大小的选择

在其他条件相同的情况下,堆空间越大,则赋值器的吞吐量越高,垃圾回收的开销越小。但在某些情况下,较小的堆可能会提升赋值器的局部性、减少转译后备缓冲区不命中的几率,进而提升赋值器吞吐量。另外,如果堆空间过大以至于物理内存无法将其容纳,则程序的执行很容易出现性能上的颠簸,特别是在垃圾回收过程中。因此,选择合适的堆空间大小,其目的通常是尽量减少程序的物理内存占用量。“足够小”的标准通常会因为运行时系统以及操作系统而产生差异,因此本节我们仅介绍自动内存管理器调整堆大小的几种策略。除了调整堆空间大小之外,减少程序所占用物理内存的策略还包括将某些页换出到磁盘(如书签回收器 [Hertz 等, 2005; Hertz, 2006])、将一些很少访问的对象保存到磁盘中 [Bond and McKinley, 2008]。

Alonso 和 Appel [1990] 设计出一种运行时调整堆大小的策略,该策略需要借助于一个“通知服务”来获取内存使用率信息,一般是通过 `vmstat` 等命令来获取。在其为 SML 设计的 Appel 式分代回收器中,每次整堆回收完成后,回收器会上报程序所需的最小内存空间、当前堆空间大小、距离上一次整堆回收的时间、自从上次回收完成以来赋值器和回收器各自占用的 CPU 时间。通知服务会据此计算出还可以为程序增加多少额外内存,程序便可据此做出相应调整。该策略的目的是在确保其他进程不出现性能颠簸的情况下尽量增大本进程的吞吐量。

208

与 Alonso 和 Appel 的策略不同, Brecht 等 [2001, 2006] 无需借助于操作系统的页相关信息便可实现对 Java 应用程序堆增长的控制。对于给定物理内存的系统(他们所研究的是 64MB 或 128MB), 他们为该系统设定一系列增量式的堆空间阈值,从 T_1 到 T_k , 每个值都是一定比例的系统物理内存。在任意时刻,进程堆空间均为 T_1 到 T_k 中的某个值。如果回收器在堆空间为 T_i 时回收所得的空间小于 $T_{i+1} - T_i$, 则系统将程序的堆空间增大到 T_{i+1} 。对于 Boehm-Demers-Weiser 回收器 [Boehm and Weiser, 1988], 由于其无法实现堆的收缩,所以该方案将仅用于控制堆的增长。该策略的问题在于,各阈值的选择不可能凭借经验来完成,同时该方案也假定其所服务的进程是系统中唯一需要关注的进程。

Cooper 等 [1992] 提出了一种将 Appel 式 SML 回收器的工作集调整到指定大小的策略,该回收器工作在 Mach 操作系统下。他们通过调整新生代大小的方式来阻止工作集的增大,除此之外他们还使用了两种 Mach 平台特有的技术。一种技术是使用较大的稀疏地址空间,

并避免将存活对象复制到更低的地址空间,从而进一步避免触达地址空间的末端。尽管该技术并不会影响堆空间大小,但它的确可以降低回收时间。另一种 Mach 平台特有的技术是回收器可以通知 Mach 页管理器直接将某一来源空间页丢弃,而不必将其换出,如果程序再次访问该页,则系统会在该页原有的地址空间中映射新的页,但页中的内容却可能是任意的内容,分配器有必要将其清零。他们在一组小型基准程序套件上进行测试,并在时间上取得四倍的提升,其中一半的提升效果都来自于堆大小的调整。但是,目标工作集合的大小仍需要用户来决定。

Yang 等 [2004] 对原版 UNIX 内核进行修改并增加了一个系统调用,应用程序可以通过该系统调用来判断在不造成系统颠簸的情况下本程序的工作集还能增大多少,或者为避免系统颠簸,其工作集应当减小多少。他们对垃圾回收器进行修改,以便利用这一信息来调整堆空间大小。他们认为,当其他进程的内存使用率发生变化时,回收器对当前进程的堆空间做适应性调整以达到最佳性能,并论证了这一操作的重要性。他们引入了程序的空间占用量 (footprint) 这一概念,即为避免程序的运行时间增大某一特定的比值 t (通常为 5% 或者 10%) 所需的物理内存页的数量。在使用垃圾回收的程序中,空间占用量取决于堆的大小,而对于复制式回收器,空间占用量还取决于整堆回收的存活率,即存活对象的总体大小。他们所得出的观察结论与 Alonso 和 Appel 并无二致,即核心关系在于当堆大小发生变化时空间占用量会发生怎样的变化。对于特定的回收算法,这一关系通常都是线性的,即两者的比值取决于选定的回收算法,例如标记-清扫回收器的比值为 1,复制式回收器的比值为 2。

Grzegorzcyk 等 [2007] 利用标准 UNIX 内核所提供的换页相关信息来调整堆空间大小。他们重点关注页换出行为、内核交换守护进程写入交换分区的页数量、缺页异常、被引用页不命中时需要从磁盘中加载的页数量、分配延迟、进程在尝试获取新页时的等待时间。这些统计信息全部都与特定进程的行为相关:换页行为所涉及的是该进程所占用的物理内存页,缺页异常与分配延迟也都是由进程本身的行为导致的。这三个指标可能预示着系统所使用的内存过多,如果进行堆空间收缩可能会令情况有所好转。他们发现分配延迟是这些因素中最好的指示因素。当回收器发现分配过程几乎不存在延迟时,它可以将堆空间在用户指定的堆大小上增大 2% (2% ~ 5% 的调整比例所达到的结果都是相似的)。当回收器感知到分内配延迟时,它可以收缩年轻代,从而将包括年轻代复制保留区在内的整个堆空间恢复到最后一次不发生分配延迟的状态。该操作最多会将年轻代的空间收缩 50%。当进程不存在内存压力时,是否引入该策略并不会对进程的性能造成(正面或负面)影响,而当进程存在内存压力时,该策略能够确保进程的性能接近于不存在内存压力时的状态,相比之下,未经调整的基准系统则会出现显著的性能下降。

到目前为止,我们所讨论的策略都只关注对单个进程所使用内存进行动态调整,这些策略相当于是给定时间点对系统内部的内存使用状态做出响应。另一方面,如果要执行的程序集合事先已知且不会改变,则通过 Hertz 等 [2009] 的策略可以预先计算出每个程序的最佳堆空间大小。该策略中,“最佳”意味着“最少的整体执行时间”,也可理解为“最高的整体吞吐量”。在执行阶段,其 Poor Richard 内存管理器会观察每个进程最近的缺页异常数以及驻留集合大小。如果缺页异常在某个时间增量内发生的次数远大于上一个时间增量内的发生次数(它们的差值超过某一阈值),则回收器会发起整堆回收以减少工作集的大小。类似地,如果驻留集合减少(即工作集合变得稀疏——译者注),也会触发整堆回收。最终系统中的各个进程会通过堆空间的充分竞争而达到最佳整体吞吐量。

Zhang 等 [2006] 所提出的动态堆大小调整机制与 Hertz 等 [2009] 的策略在思想上是类似的, 但在其方案中, 进程自身可以检测每次回收过程中缺页异常的数量并自主调整堆大小, 从而无需将这一机制构建在回收器中。与我们所介绍的其他机制不同, 他们假设开发者或多或少都能够确定程序执行的各个阶段, 并且会在阶段发生变化时强制呼起垃圾回收。他们表示, 相比于将堆空间大小固定的策略, 动态自适应堆调整策略可以显著提升程序的性能。

11.12 需要考虑的问题

内存分配接口给系统的实现者提出了一系列必须回答的问题。某些问题的答案取决于编程语言的特定语义, 或者运行环境所要求的并发级别 (即对象是否会从其诞生的线程“泄漏”)。其他问题的答案则可以具有一定的灵活性, 系统的实现者可能会以提升运行时系统的性能或者鲁棒性为目的来给出自己的解答。

我们需要考虑分配过程以及初始化过程必须满足哪些要求: 编程语言运行时的工作是仅需要分配足够大小的空间, 还是在对象可以使用之前必须初始化其头部的某些域? 是否需要为对象内部的每个域都提供初始值? 对象的字节对齐要求是什么? 运行时系统是否需要区分不同类别 (kind) 的对象 (例如将数组与其他对象区分)? 是否应该对不包含指针的对象进行区别对待? 回收器无需对不包含指针的对象进行扫描, 因而将其区分对待有助于提升追踪性能。事实证明, 避免对此类对象进行扫描可以显著提升保守式回收器的性能。

我们通常会仔细考虑分配过程中的哪些代码序列应当内联。我们通常应将快速路径而非慢速路径内联, 所谓快速路径即所需工作量最小的分配方式, 而慢速路径则通常需要从更底层的内存分配器中获取空间, 甚至有可能呼起垃圾回收器。但需注意的是, 过度内联会增大代码体积, 甚至有可能产生负面效应。为提升效率, 系统甚至可以将某一寄存器专门用于特殊用途, 例如顺序分配中的阶跃指针。但在寄存器数量较少的平台上, 这一策略可能会给寄存器分配器造成较大压力。

[210]

在某些语言中, 出于安全性考虑或者调试原因, 运行时系统可能需要将内存清零。系统可以在分配对象时将其空间清零, 但使用高度优化的库函数对大块内存进行清零更加高效。在赋值器即将使用某一内存之前将内存清零可以获取最佳的高速缓存性能, 而如果在内存被释放之后立即将其清零则有助于调试 (在对象占用的空间中写入特殊值可能效果更佳)。

回收器需要通过指针查找来确定对象的可达性。运行时系统提供给回收器的指针信息可以是精确的, 也可以是保守的, 除此之外还可以将两者结合, 即线程栈中的指针信息是保守的, 而堆中的指针信息则是精确的。保守式指针查找的实现较为简单, 但它却增加了内存泄漏的风险, 同时其性能会比类型精确的回收策略要差。栈中指针的查找会给类型精确回收器的设计带来一定挑战, 特别是当栈中包含混合帧类型 (经优化的与未经优化的子过程、本地代码帧、桥接 (bridging) 帧) 时。但在另一方面, 通过栈扫描的方式查找指针限制了回收算法的选择[⊖], 因为此时直接被栈引用的对象将无法移动。

系统通常会使用栈映射来确定某一返回地址位于哪个函数内部。多态函数以及特定的语言设计 (例如 Java 的 jsr 字节码) 会导致栈映射的复杂化。实现者还必须决定在何时生成栈映射以及何时可以使用使用栈映射: 是应当提前生成, 还是应当采用懒惰生成策略以节省空间? 每个栈映射是否只在特定的安全回收点有效? 栈映射可能会很大, 因此如何才能将其压

⊖ 即回收器必须使用保守式指针查找策略来查找栈上指针。——译者注

缩（特别是当要求栈映射对于每条指令都有效时）？栈扫描的实现还与一些其他问题相关，即栈扫描过程是完整的、原子性的，还是增量式的。尽管增量栈扫描更加复杂，但它具有两个优点：第一，每个增量中的扫描工作量可以得到限制（这对于实时回收器十分重要）；第二，如果可以获知栈中的哪些部分在上次栈扫描之后未发生变化，则我们可以降低回收器需要完成的工作量。

编程语言的特定语义以及编译器优化会进一步提升指针查找的复杂度。如何处理内部指针与派生指针？编程语言可能允许赋值器访问托管环境之外的对象，它们通常是使用 C/C++ 开发的，除此之外，每种编程语言的输入/输出都需要与操作系统进行交互。因此，运行时系统就必须确保对象在被外部代码使用时不会得到回收。相关解决方案通常包括：将此类对象钉住或者要求外部代码使用句柄来访问它们。

某些系统可能允许垃圾回收过程发生在程序的任意位置，但如果可以将垃圾回收的发生位置限定在特定的安全回收点，一般可以简化系统的设计。安全回收点通常包括分配函数、后退分支、函数的入口以及返回位置。有多种方式可以将线程挂起在安全回收点：一种策略是通过一个全局旗标来反映当前是否需要垃圾回收，每个线程通过轮询的方式来检测该旗标；另一种策略是对正在执行的线程的代码“打补丁”，并引导其挂起在下一个安全回收点。回收器和赋值器线程之间的握手方式包括令线程检查某一线程局部变量、在被挂起线程保存的上下文中设置处理器条件码、劫持函数返回地址、通过操作系统信号。

许多回收算法都要求赋值器在执行过程中记录回收相关指针，为此，研究者们设计并实现了多种策略，以探测并记录回收相关指针。屏障操作通常十分普遍，因而尽量将其引入的开销最小化便显得十分重要。屏障既可以是由编译器插入在指针加载或者存储操作之前的一小段代码序列，也可以借助于操作系统来实现，例如页保护陷阱。屏障的设计依然需要对多种因素进行权衡，包括赋值器开销与回收器开销之间的平衡、屏障的精度与速度之间的平衡。我们通常更倾向于让执行频率相对较低的回收相关操作（例如查找程序的根）来承担更多工作，从而尽量保证执行频率更高的赋值器操作（例如针对堆中对象的写操作）的性能。写屏障可能会导致指针写操作的指令翻倍甚至更多，但这一开销通常会被高速缓存不命中所掩盖。

[211]

对回收相关指针的记录应当达到何种精度？将回收无关指针过滤掉后精度可能更高，但无条件记录被修改指针给赋值器引入的开销更小。记忆集的实现方式决定了记录的精度。指针过滤代码应当内联到何种程度？这需要精细地进行调整。指针位置的记录应当达到何种粒度？我们是应当记录被修改的域，还是其所在的对象，还是其所在的卡或者页？是否允许记忆集包含重复对象？数组与非数组是否应以相同的方式处理？

应当使用哪种数据结构来记录回收相关指针？哈希表、顺序存储缓冲区、卡表，还是综合这些数据结构？所选择的数据结构分别会给赋值器和回收器带来怎样的开销？如何安全且高效地处理数据结构可能发生的溢出？卡表是一种不甚精确的记录方式，回收器在回收阶段必须对其进行扫描并找出脏卡，然后再进一步找出包含回收相关指针的对象。这一过程中有几个性能相关问题需要注意：卡的大小应该设置为多大？卡表通常比较稀疏，如何才能加速其中脏卡的查找？是否需要使用两级卡表？是否可以对卡进行汇总（例如当卡中只包含一个被修改域或者对象时）？当回收器找到脏卡时，它必须定位出该卡中的第一个对象，但是该对象的起始地址却可能位于更靠前的卡中，因此我们需要借助于跨越映射来查找跨越多个卡的对象。卡标记会与多处理器高速缓存一致性协议产生怎样的相互影响？如果两个处理器不

断对位于同一个卡上的对象进行写操作，它们都需要以独占方式访问卡所在的高速缓存行，这样是否会产生高速缓存冲突？在实际环境中，这是否可能会成为问题？

在使用虚拟内存的系统中，基于垃圾回收的应用应当能够根据真正可用的内存进行自适应调整，这一点十分重要。与非托管程序不同，垃圾回收系统可以通过调整堆大小的方式来更好地适应可用内存。回收器可以利用操作系统所提供的哪些事件以及统计信息来调整堆空间大小？这些事件或者信息中的哪些最高效？如何才能设计出一种较好的堆增长策略？如果堆可以收缩，那么如何进行收缩更加合理？多个基于垃圾回收的程序彼此之间如何进行协作，才能达到较高的整体吞吐量？

综上所述，垃圾回收的许多细节问题都比较琐碎，但它们却会在性能方面显著地影响系统的设计和实现。后续章节中，我们将看到本章所描述的技术在垃圾回收算法中的具体应用。

特定语言相关内容

许多编程语言都具备垃圾回收能力，因此开发者们自然希望将这一能力拓展到自动内存管理之外的更多场景。在这一思想的指导下，研究者们发展出了多种在应用程序和回收器之间进行交互的方式，这些交互方式拓展了编程语言的基本内存管理语义。例如，应用程序可能希望在对象即将被回收或者已经被回收后得到一些通知，或者执行一些额外操作，我们将在 12.1 节讨论这种终结（finalisation）机制。另外，某些情况下我们不希望某个引用会对其目标对象的存活性产生影响，我们将在 12.2 节讨论这种弱指针（weak pointer）机制。

12.1 终结

使用垃圾回收器进行自动内存管理可以为大多数对象提供合适的内存管理语义。但是，如果托管对象引用了托管范围之外的其他对象，自动垃圾回收器将无能为力，进而可能导致资源的泄漏。一个典型的案例是程序所打开的文件：操作系统接口通常会以一个被称为“文件描述符”的小整数来表示一个文件，该接口限制了给定进程在同一时刻可以打开的最大文件数量。编程语言通常会将每个打开的文件封装成对象，以供开发者管理文件流。大多数情况下，当程序完成某一文件流的处理之后，开发者可以通过运行时系统来关闭文件流，后者会调用操作系统接口来关闭对应的文件描述符，以便将其复用。

但是，如果文件流被程序中的多个组件共享，运行时系统将很难确定在何时所有组件都能完成文件流的处理。如果使用某一文件流的每个组件在使用完毕后都将指向文件流的引用置空，则当该文件流对象不再被任何组件引用时，回收器（终究）会探测到这一情况。图 12.1 展示了这一状态，我们或许可以借助于回收器来实现文件描述符的关闭。

为达到这一目的，我们需要在给定对象不可达之后执行用户自定义的行为——更加确切地讲，应该是对象不再从赋值器可达时。我们将这一过程称为终结。典型的终结机制允许用户指定一段代码，即终结方法（finaliser），回收器在判定特定对象不可达时将调用该方法。终结机制的典型实现方式是由运行时系统维护一张特殊的终结表，该表中所记录的是包含（由开发者指定的）终结方法的对象。赋值器无法访问到该表但回收器

可以。我们将从终结表可达但从赋值器不可达的对象称为终结可达（finaliser-reachable）对象。图 12.2 展示的情况与图 12.1 一致，唯一的不同之处在于其增加了终结方法。由于应用

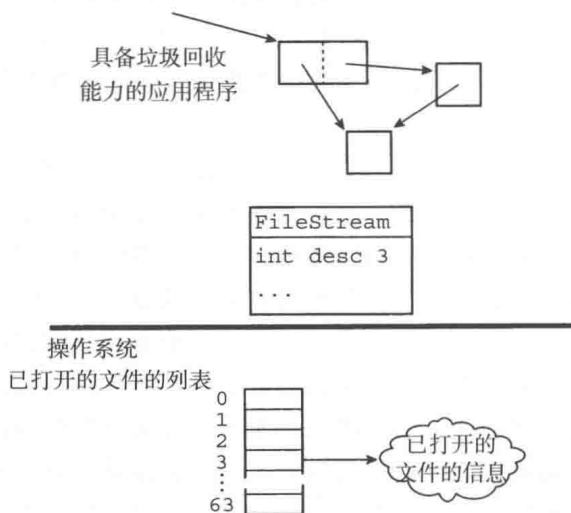


图 12.1 资源泄漏。FileStream 对象已经不可达，但其所对应的文件描述符尚未关闭

程序可能提前关闭文件，所以终结方法在关闭文件描述符之前应当进行条件判断。

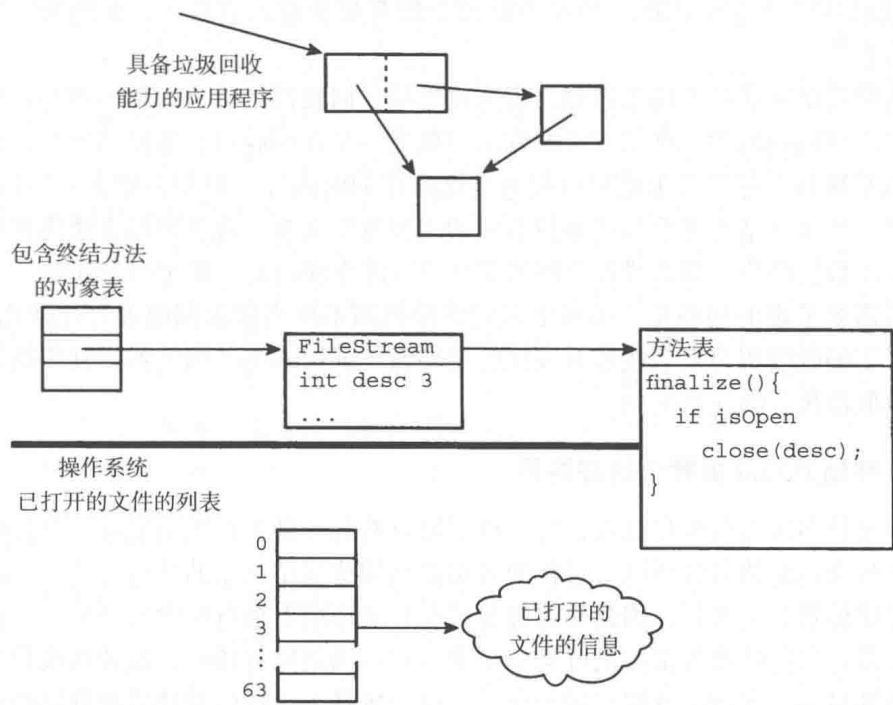


图 12.2 使用终结方法来释放资源，不可达的 `FileStream` 对象通过终结方法来关闭描述符

在引用计数系统中，回收器可以在释放对象之前查找终结表，并判断该对象是否需要终结，如果需要，则回收器执行终结方法，并将其从终结表中移除。类似地，追踪式回收系统也可以在追踪阶段完成之后检查终结表，找出其中的未标记对象并执行终结方法，然后将其从终结表中移除。

终结机制有多种不同的实现策略，它们之间会存在一些细微的差别，接下来我们将介绍终结机制的实现方式以及可能遇到的问题。

12.1.1 何时调用终结方法

终结方法应在何时调用？一种实现策略是：一旦回收器发现需要终结的不可达对象，便立即调用其终结方法。但该策略的问题在于，回收的中间状态可能无法执行一般用户代码，例如此时可能无法进行新对象的分配。因此，大多数终结机制都在回收过程之后才调用终结方法。回收器可以使用队列来组织待终结对象。为避免在回收过程中分配这一队列，回收器可以将终结表划分为两个分区，一个分区用于容纳待终结对象，另一个则用于记录包含终结方法但尚未入队（即依然存活——译者注）的对象。当回收器需要将某一对象加入待终结队列时，它只需要将该对象移动到终结表的待终结分区。一种简单但稍显低效的方案是为终结表中的每个对象关联一个待终结位，此时回收器就需要通过扫描终结表来找到待终结对象。为避免这一扫描过程，我们可以将表中的待终结对象放置在一起，此时回收器便可通过换位的方式来添加新的待终结对象。

终结方法通常会影响线程之间的共享状态。我们不能因为待终结对象即将消亡而将终结方法的操作范围限定在对象内部。例如，终结方法可能需要访问全局数据以实现某一共享资源的释放，该操作很可能需要加锁。这也是回收器不能在回收阶段调用终结方法的另一个原

因, 即可能引发死锁。更加糟糕的一种情况是, 如果运行时系统提供了可重入锁 (即允许线程重复获取其已经持有的锁), 终结方法便会绕开死锁进入临界区, 并悄无声息地破坏应用程序的状态^①。

即使将终结方法的调用推迟到回收完成之后, 回收器依然会面临一些与在回收过程中调用终结方法相同的问题。终结方法的调用时机之一是在回收过程刚刚结束时, 此时赋值器线程尚未恢复执行。尽管该策略可以提升终结操作的时效性, 但却会增大赋值器的停顿时间。除此之外, 如果终结方法会与其他持有锁的线程发生交互, 或者终结方法需要通过加锁的方式争用全局数据结构, 那么线程之间的交互便可能出现問題, 甚至引发死锁。

最后需要考虑的问题是, 编程语言的终结机制不应当限制回收器可能使用的回收技术。例如, 对于回收线程与赋值线程并发执行的即时 (on-the-fly) 回收器, 其终结方法的调用会发生在赋值器执行的任意时刻。

12.1.2 终结方法应由哪个线程调用

[215]

对于支持多线程的编程语言, 终结机制最自然的一种实现策略是在后台运行一个终结线程, 该线程会在赋值器线程执行过程中异步地调用待终结对象的终结方法。此时终结方法便可能会与赋值器并发执行, 因此必须确保其在并发环境下执行时的安全性。一种特别需要关注的情形是: 当终结线程正在执行类型 *T* 某一实例的终结方法时, 赋值器线程有可能在相同时刻执行其另一个实例的分配与初始化。在这一场景下, 任何对共享数据结构的操作都必须以同步方式执行^②。

对于只支持单线程的编程语言, 由哪个线程来调用终结方法显然不会是一个问题, 但此时的問題将主要集中在何时调用终结方法上。我们前面已经指出了确定执行时机的困难所在, 因此在单线程环境下, 唯一可行且安全的方案是待终结对象进行排队, 并在显式控制之下调用其终结方法。而在多线程系统中, 正如我们前面所提到的, 最佳的解决方案是使用独立的终结线程来执行终结方法, 从而避免锁相关问题。

12.1.3 是否允许终结方法彼此之间的并发

如果在大型并发应用中使用终结机制, 则可能需要更多的终结线程来确保可伸缩性。因此从编程语言的设计角度来看, 不仅应当允许终结线程之间并发执行, 而且还应当允许终结线程和赋值器线程之间并发执行。因此, 开发者通常必须谨慎地设计终结函数来应对可能出现的并发情况, 只要能够确保终结线程与赋值器线程并发执行时的安全性, 终结线程彼此之间的并发便不会存在更多问题。

12.1.4 是否允许终结方法访问不可达对象

在许多场景下, 用户都希望终结方法能够访问待回收对象。在图 12.2 所描述的文件流案例中, 我们可以很自然地将操作系统文件描述符 (一个小整数) 放在文件流对象的某个域

① 为避免这一情况的出现, Java 使用专门的终结线程来调用终结方法。该线程在调用终结方法之前不会持有任何锁, 因此终结线程必然不可能获取到待终结对象已经持有的锁。实际应用中, 将终结工作指派给专门的终结线程并与一般线程进行区分, 这通常是一种比较好的做法。

② Java 通过一种特殊的规则来避免这一情况: 如果对象的终结方法可能引发同步操作, 则不论该对象是否持有锁, 回收器都会将其当作赋值器可达对象, 这一策略可以避免对终结方法中同步操作的限制。

中，此时终结方法最简单的实现方案便是读取该域，并调用操作系统接口来关闭文件（在此之前可能需要刷新缓冲区中尚未输出的数据）。但是，如果终结方法无法访问对象，而只能执行一小段未关联任何数据的代码，则终结机制的效用将大打折扣——终结方法需要基于特定的上下文进行工作。这一上下文信息在函数式语言中可能是一个闭包，在面向对象语言中可能是一个对象。因此终结机制需要为终结方法提供特定的参数。

总的来说，允许终结方法访问即将终结的对象的灵活性更高。假设终结方法是在回收完成后执行，这一策略意味着待终结队列中的对象必须存活到回收结束之后。由于终结方法可能访问任何从其上下文可达的对象，所以回收器必须避免回收所有从待终结对象可达的对象，这便导致追踪式回收器必须额外引入两个阶段：第一阶段找出所有待终结对象，第二阶段对待终结对象进行追踪并保留所有从待终结队列可达的对象。基于引用计数的回收器可以在对象加入到待终结队列时增加其引用计数，相当于是待终结队列引用了该对象并为其贡献了引用计数。一旦终结线程将对象从待终结队列移除并调用其终结方法，其引用计数便会降低为零，进而得到回收，但在此之前，所有从该对象可达的对象都不会得到回收。

12.1.5 何时回收已终结对象

终结方法会持有待终结对象的引用，这意味着其可能会将该引用添加到某一全局数据结构中。这一现象被称为复活（resurrection）。尽管复活机制本身并不存在问题，但得到复活的对象通常不会再次终结，这可能会令开发者感到诧异。其原因在于，系统通常难以探测到引发对象复活的写操作，且将对象加入终结表的操作通常是对象分配以及初始化过程的一部分。例如 Java 便保证对象不会被终结超过一次。对于支持以更加动态的方式建立终结任务的编程语言，开发者可以请求运行时系统对某个已复活对象再次进行终结，因为编程语言允许对任何对象执行这样的操作。

[216]

如果已终结对象未被复活，则其将在下一轮回收过程中得到回收。对于使用分区策略的回收器（例如分代回收器），已终结对象可能会驻留在某个回收间隔较长的空间内，因此终结方法可能会显著延长对象的物理寿命。

12.1.6 终结方法执行出错时应当如何处理

如果应用程序以同步的方式来执行终结操作，则开发者很容易对其中的终结操作进行包装以便捕获其所返回的错误或者抛出的异常。但如果终结方法以异步的方式执行，则最好的方法是捕获异常并将其记录，同时将其交由应用程序在合适的时间处理。此处的相关内容更多地涉及软件工程学的相关内容，超出了垃圾回收算法的讨论范畴。

12.1.7 终结操作是否需要遵从某种顺序

终结顺序与应用程序密切相关。考虑图 12.3 所示的场景，`BufferedStream` 类的实例引用了一个类型为 `FileStream` 的对象，后者持有一个已经打开的操作系统文件描述符。两个对象都需要终结，但是在 `FileStream` 对象关闭文件描述符之前，应用程序必须确保 `BufferedStream` 实例先将缓冲区中的数据刷新到文件中^①。

① 我们还需要注意另一种更加微妙的情况：除非可以确定图中的 `FileStream` 对象仅被 `BufferdStream` 对象引用，否则在后者完成终结之后，回收器依然不能终结 `FileStream` 对象。但这样一来，`FileStream` 对象中的文件描述符要到两个回收周期之后才能得到关闭。

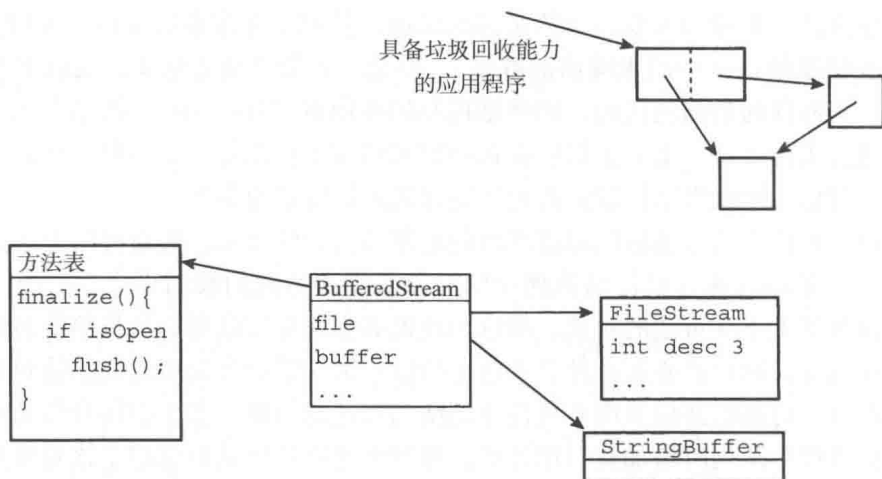


图 12.3 对象终结顺序。不可达的 `BufferedStream` 实例必须先于不可达的 `FileStream` 实例执行终结

对于类似于图 12.3 的分层设计模型，合理的终结语义显然应当是从上到下逐层进行终结。由于下层对象从上层对象可达，所以对象之间自动依照这种顺序进行终结是有可能的。需要注意的是，如果我们限制了对对象的终结顺序，则终结过程的最终完成可能需要很久，因为每次回收仅可以终结位于某一层级的对象。也就是说，每次回收过程中我们仅可以终结那些不从其他待终结对象可达的对象。

这一策略存在一个显著的缺陷：它无法处理由多个待终结对象组成的环。但是这一情况通常极少发生，因而确保对象之间依照可达性顺序来进行终结通常更加简单有效，也就是说，如果对象 B 从对象 A 可达，则系统应当先终结对象 A。

一旦待终结对象组成环，开发者必须进行手工干预。诸如弱引用（参见第 12.2 节）等机制虽然较为复杂，但对解决这一问题也可能有所帮助。正如 Boehm[2003] 所指出的，通用的解决方案应当是在设计时将需要终结的域从对象中拆分，进而打破待终结对象可能构成的环。例如在图 12.4a 中，对象 A 和 B 都存在终结方法且彼此相互引用。为避免终结过程中环的出现，我们可以将 B 拆分为 B 和 B'，其中 B 不包含终结方法而 B' 包含（参见图 12.4b），此时尽管对象 A 和 B 依然相互引用，但重要的是 B' 并未引用 A。此时回收器便可依照可达顺序先终结 A，再终结 B'。

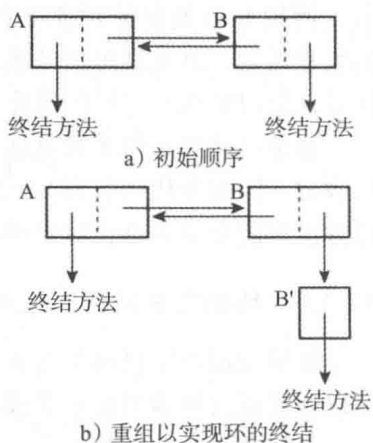


图 12.4 对环状对象图进行重组以确保终结顺序

12.1.8 终结过程中的竞争问题

即使终结操作的执行顺序并无特定的顺序要求，依然存在一种微妙的竞争问题，这一问题导致直接使用终结过程会存在一些十分隐晦的错误 [Boehm, 2003]。重新考虑图 12.2 所示的 `FileStream` 案例。假设在赋值器最后一次向文件中写入数据的过程中，`FileStream` 对象的 `writeData` 方法会先获取文件描述符，然后再执行 `write` 系统调用向其中写入数据。由于在 `write` 系统调用之后 `FileStream` 对象便会死亡，所以编译器可能会进行优化，即在执行 `write` 系统调用之前 `FileStream` 对象便已经不可达。如果回收过程发生在 `write` 操作的

执行过程中, 则 `FileStream` 对象的终结方法可能会在 `write` 调用真正引发操作系统的写入之前便将描述符关闭。这一问题较为棘手, 且 Boehm 根据经验认为此类错误是普遍存在的, 但是由于可能出错的时间窗通常较短, 所以问题很少暴露出来。

在 Java 语言中, 这一问题的解决方案我们在前面已经提到过, 即一旦对象的锁被持有, 则回收器将其判定为存活, 且该对象的终结方法只能同步执行。但更加通用的避免竞争的解决方案是强制编译器将 `FileStream` 对象的引用保持更长时间, 即把该对象的引用传递给稍后的某一调用 (该调用可以不执行任何操作), 且编译器不会将这一调用优化掉。.NET 框架提供了这一能力, 例如 C# 中的 `GC.KeepAlive` 函数, 但 Java 目前并未提供类似的调用。

218

12.1.9 终结方法与锁

Boehm[2003] 提出, 终结方法的目的通常是通过更新某些全局数据结构来释放与不可达对象相关的资源, 所以此类数据结构是全局的, 因而对它们的访问通常需要引入一些同步机制。关闭已打开的文件或者其他软件组件 (此处是指操作系统) 句柄的操作通常会引发隐式同步, 但对程序数据结构的更新则必须引入显式同步, 因为对于程序中大多数代码而言, 终结方法的执行是异步的。

开发者可以通过两种策略来处理这一情况。一种策略是令所有针对全局数据结构的访问都使用同步操作, 即使是在单线程情况下也必须如此 (因为终结方法可能会在全局数据结构的某一中间状态对其进行操作)。这要求编程语言不得将包含终结方法的、明显私有的对象所需的同步操作省略。另一种策略是回收器仅将待终结对象排队, 但不执行真正终结操作。某些编程语言的内置终结机制本身就使用这种排队策略, 但也有一些并非如此, 此时开发者便需要手工将待终结对象添加到自定义终结队列中。除此之外, 开发者还需要在适当的 (安全) 位置增加代码来处理终结队列中的对象。由于终结方法的执行可能会导致新的待终结对象被添加到终结队列中, 所以队列处理代码应当一直处理到整个队列变空为止, 如果处理过程中有一些重要的资源需要立即释放, 处理代码可以强制发起回收。算法 12.1 即为这一操作的伪代码实现。上文提到, 执行这一算法的线程不应当持有任何待终结对象的锁, 这便限制了终结队列处理代码可以安全执行的位置。

算法 12.1 终结队列的处理

```
1 process_finalisation_queue():  
2     while not isEmpty(Queue)  
3         while not isEmpty(Queue)  
4             obj ← remove(Queue)  
5             obj.finalize()  
6         if desired /* 某种适当的条件 */  
7             collect()
```

使用该方案唯一需要注意的是, 开发者需要选择合适的位置来执行终结队列处理代码。除了必须手工实现所有的终结逻辑之外, 开发者还应当注意避免在共享对象的中间状态对其进行操作。仅使用锁可能是不够的, 因为调用终结方法的线程可能已经持有了锁, 所以临界区重入的现象有可能发生。Java 编程规范指出, 系统只有在线程在不持有任何用户可见锁的情况下, 才会调用终结方法, 其原因正在于此。

12.1.10 特定语言的终结机制

Java。Object 类位于 Java 类继承体系的最上层, 该类提供了一个名为 `finalize` 的方法,

[219]

但该方法并不做任何事情。子类可以通过重载该方法的方式来请求终结。Java 并不保证终结顺序, 且其在并发方面唯一可以提供的保障是终结方法仅会在不持有任何用户可见锁的上下文中执行。这也意味着终结方法可能在多个线程中并发执行, 尽管手册并未对这一情况进行说明。如果 `finalize` 方法抛出异常, Java 系统会将其忽略并继续执行。如果待终结对象并未复活, 则其将在后续的回收过程中得到回收。Java 使用 `java.lang.ref` API 支持用户控制的终结机制, 这将在 12.2 节进行描述。

Lisp. Liquid Common Lisp 提供了一种被称为终结队列 (finalisation queue) 的对象。开发者可以将普通对象注册到一个或者多个终结队列中。当已注册对象不可达时, 回收器会将其添加到其所注册的终结队列中。开发者可以从任意一个终结队列中获取对象并执行其需要的操作。假设对象 A 和对象 B 均已注册到终结队列中, 且对象 B 从对象 A 可达 (但反之不可达), 如果它们在同一轮回收中不可达, 回收器可以确保对象 A 先于对象 B 加入终结队列。也就是说, 回收器可以确保无环对象图的终结顺序。Liquid Common Lisp 中的终结队列与 Dybvig 等 [1993] 所描述的守护者 (guardian) 策略类似。

CLisp 提供了一种更加简单的机制, 它允许开发者要求回收器在发现给定对象 O 不可达时调用给定的函数 f 。在这一场景下, 函数 f 不应当引用对象 O, 否则对象 O 将保持可达, 系统将永远不会将其终结。由于对象 O 是作为参数传递给函数 f 的, 所以系统允许对象复活。函数 f 甚至可以再次注册对象 O, 即对象 O 可以被终结多次。这一基本策略的一个变种允许开发者为对象 O 以及函数 f 指定守护者 G: 当对象 O 不可达时, 只有在守护者 G 依然可达的情况下, 系统才会调用函数 f 。如果守护者 G 不可达, 则系统依然会回收对象 O 但是不会调用函数 f 。对于 Dybvig 等 [1993] 所描述的这种守护者策略, 其实现方式之一是在函数 f 中将对象 O 加入到守护者 G 的内部队列中。

C++. C++ 语言提供了析构函数 (destructors) 来处理对象的销毁, 其相当于初始化新对象的构造函数的逆操作。大多数析构函数的主要目的是显式释放内存并将其归还给分配器。除此之外, 开发者还可以在析构函数中添加任意代码, 因而其可以完成文件关闭等任务。开发者也可将析构函数作为钩子 (hook) 来实现非环状共享对象的引用计数回收。实际上, C++ 模板允许通过一种通用的智能指针机制来实现引用计数。析构函数中的大多数工作通常都与内存释放有关, 这正是垃圾回收器所要处理的任务, 因此对 C++ 来说, 几乎无需引入额外的终结机制。析构函数中的内存释放通常相对安全且直接, 这不只是因为它不会引入用户可见的锁那么简单。但是, 一旦开发者需要面临真正的终结场景, 则前面提到的所有问题, 包括锁相关处理、终结方法的调用顺序等, 都将重新出现并需要开发者去解决, 要正确处理这些问题通常较为困难。

[220]

.NET. 在 C++ 现有的析构函数之外, .NET framework 为 C、C++ 及 framework 所支持的其他语言增加了终结方法。析构函数的调用是确定性的, 它是由编译器生成的代码所调用的, 其目的在于当程序离开对象作用域时执行必要的对象清理。析构函数可能会调用其他对象的析构函数, 但是所有的析构函数都与托管资源 (即 .NET 运行时系统控制的资源, 主要是内存) 的释放有关。而终结方法的目的则在于显式回收非托管资源, 例如打开的文件句柄等。如果某类对象需要终结, 则对于编译器生成代码显式回收对象的场景, 析构函数需要调用终结方法。如果对象最终得到隐式回收, 则终结方法的调用最终是由回收器来完成的, 此时析构函数将不会被调用。不论如何, .NET framework 中的终结机制与 Java 十分类似, 但其最终形态却混合了 C++ 的析构函数以及某些与 Java 十分类似的终结机制, 其终结方法的调用既可能是同步的, 也可能是异步的。

12.1.11 进一步的研究

许多语言在很早之前就已经支持终结机制，并且不同的语言会根据自身情况引入一些特殊机制。对于终结机制的相关问题及其不同设计方式，系统性的、跨语言的分析在最近的文献中出现的越来越多，例如在 Hudson [1991] 以及 Hayes [1992] 的文献中。Boehm[2003] 对终结语义以及其中的某些棘手问题进行了深入分析，该文献也是本节相关内容的主要来源。

12.2 弱引用

垃圾回收机制通过指针链的可达性来判定哪些对象需要保留、哪些对象需要回收。对于自动内存管理而言这是一种合理的策略，但其在某些场景下依然可能遇到问题。

例如，某些编译器会令同名变量（例如 `xyz`）指向相同的名称字符串实例，此时如果要比较两个变量的名称是否相等，只需比较它们指向自身名称字符串的指针是否相等。为达到这一目的，编译器需要通过一张表来记录所有已经使用过的变量名。该表中的字符串即为变量名的规范实例（canonical instance），因此该表在某些情况下也被称为规范化表（canonicalisation tables）。如果某个变量名在运行期间不再使用（即该变量名不再对应任何数据结构），但其所对应的规范实例将依然存在。运行时系统或许可以将仅被规范化表所引用的字符串回收，但可靠地探测到这一情况却比较困难。

弱引用（wak reference）（也称弱指针（weak pointer））可以较好地解决这一问题。如果从根出发，经由一系列由强引用（strong references）构成的指针链可以到达某一对象，则称该对象强可达（strongly reachable）。只要对象依旧强可达，则指向该对象的弱引用便可一直保持其引用关系。但是，一旦从根出发到达该对象的任意一条指针链都包含至少一个弱引用，则回收器可以将该对象回收，并将所有直接引用该对象的弱引用设置为空。我们将此类对象称为弱可达（weakly-reachable）。我们即将看到，回收器在回收弱可达对象时还会执行一些额外动作，例如通知赋值器某一弱引用已被设置为空。

在上文提到的使用规范化表来维护变量名的案例中，如果将从规范化表到变量名的引用设置为弱引用，则一旦用于表示某一变量名的字符串不存在强引用，回收器便可回收该字符串并将其在规范化表中的弱引用设置为空。需要注意的是，规范化表的设计也需要将这一因素考虑在内，即程序可能偶尔需要清理规范化表，或者值得进行清理。例如，如果规范化表使用哈希表的方式实现，且哈希表里每个桶中均维护有一条链表，则已经死亡的弱引用（即已经被置空的弱引用）可能会占用链表中的部分节点，因此我们可能需要偶尔将这些引用从哈希表中清除。这同时也说明了通知机制的作用：我们可以利用通知来触发清除逻辑。

后续我们将提供更加通用的弱引用定义，其中包含多种不同强度的引用类型，同时我们也将说明回收器应当如何支持这些引用。但在此之前，我们仅考虑强引用和弱引用的实现方式。首先考虑追踪式回收器中的场景。为支持弱引用，回收器在首轮遍历过程中应当避免对弱引用的目标进行追踪，但需对其进行记录以便第二次遍历。第一次遍历完成之后，回收器将找出所有经强引用指针链可达的对象，即强可达对象。在第二次遍历过程中，回收器将检查第一轮遍历过程中发现并记录的弱引用：如果其所引用的对象依旧强可达，则回收器保留该弱引用（复制式回收器还需将其更新到目标对象的最新副本）。否则，回收器将把该弱引用设置为空，从而确保其目标对象不再可达。第二轮遍历完成后，回收器便可回收所有不可达对象。

回收器必须能够识别弱引用。可以在引用中通过一位来将其标记为弱引用，例如对于依

照字节定址的、按照字来对齐的机器，指针的低两位必然为零，我们便可将其中的一位设置为 1 来表示弱引用。该方案的缺陷在于每个解引用操作都要先清空其最低位以避免当前引用是弱引用。如果弱引用仅用于编程语言的特定受限场景，这一开销或许可以接受。某些语言及其实现可能会使用带标签值 (tagged value) (参见 11.2 节——译者注) 来实现这一策略，但这样一来弱引用又会引入一种新的标签类型。该方案的另一个缺陷在于，回收器必须找到所有指向待回收对象的弱引用并将其置空，因此回收器要么必须对根和堆进行第二次遍历，要么必须记录第一次遍历过程中发现的所有弱引用。

与使用低位作为标签值的方案相对，另一种方案是使用高位作为标签值，并对整个堆进行二次映射。此时堆中的每一页都会在虚拟内存中的两个位置出现，一个位置是其原本的地址，另一个位置则是较高的 (不同) 地址。两个地址的唯一不同之处在于高位中用于区分弱引用的位有所不同。该方案可以避免在使用指针之前先进行掩码操作，且其检测弱指针的方式也十分简单高效。但其缺点在于，可用地址空间少了一半，这在较小的地址空间中将成为问题。

最通用的实现方案可能是使用间接方式，即提供专门的弱对象来持有弱引用。弱对象方案的不足之处在于其透明性不足：回收器和赋值器均需要一次显式解引用操作才能访问弱引用的目标对象，从而引入了间接访问开销。除此之外，如果我们需要在某一对象中引入弱引用，还需额外分配一个弱对象。但幸运的是，弱对象的特殊性只需要分配器和回收器关注即可，对于用户代码而言，它们与普通对象并无二致。系统可以在对象头部设置一个特殊的位来区分弱对象。另外，如果对象包含一些用户自定义的追踪方法，则弱引用仍需要进行特殊处理。

开发者如何才能获取一个弱引用 (弱对象)？在使用弱引用的真实场景下，系统必须提供一个原语，开发者可以通过该原语从对象 O 的强引用中获取其弱引用。如果使用弱对象来封装弱引用，弱对象类型通常需要提供提供一个构造函数，该函数可以从给定对象 O 的强引用中创建一个新的弱引用。系统甚至可以允许开发者改变弱对象中的引用域。

12.2.1 其他动因

[222]

弱引用可以用于解决某些编程问题，或者可以提供更加简单高效的解决方案。规范化表只是使用案例之一，另一个案例是用于管理在必要时可以恢复或者重建的数据缓存。此类缓存的目的在于达到时间和空间上的某种平衡，但如何控制缓存大小却是一个比较困难的问题：如果空间足够大，则缓存可以更大，但应用程序如何才能确定可用空间是否足够充裕？如果空间限制发生动态变化应当如何处理？回收器能够掌握可用空间的状态，因而可以将这些任务交由回收器来完成。也就是说，我们可以使用弱引用来管理缓存，回收器在发现缓存不再强可达之后可以将弱引用设置为空。此时回收器便可根据空间使用情况来调整缓存。

回收器还可以将对象从强可达到弱可达的状态变化通知给应用程序，后者可以在该对象得到回收之前执行适当的操作。该场景属于第 12.1 节所介绍的终结机制的推广。在其他条件之外，以适当的方式来组织弱引用有助于程序更好地控制对象终结方法的调用顺序。

12.2.2 对不同强度指针的支持

在强引用之外，弱引用可以泛化成多种不同强度的弱指针，它们可以处理上文所描述的多种问题。以引用强度为顺序的回收可以为每种强度级别关联一个正整数。对于给定的整数 $\alpha > 0$ ，如果从根出发存在一条指针链可以到达某一对象，且该指针链中的所有指针强度均不小于 α ，则称该对象为 α^* 可达。可达 (没有 “*” 号) 意味着对象 α^* 可达但并非 $(\alpha+1)^*$

可达。如果某一对象的所有可达路径都存在至少一个强度为 α 的指针，且至少有一条路径不包含强度小于 α 的指针，则该对象为 α 可达。描述强度的数字可以是任意的，因为我们只需要用其来表述强度的相对顺序，在后续表述中我们将用名称代替数字来描述强度的级别。

每种强度的引用通常会与回收器的特定行为相关联。在支持多种不同强度的弱引用的编程语言中，最知名的当属 Java，其提供的引用类型从最强到最弱可以分为以下几种^①：

强引用 (strong reference)：普通的引用，具有最高的引用强度。回收器永远不会将强引用置空。

软引用 (soft reference)：回收器可以根据当前的空间使用率来判定是否需要将软引用置空。如果 Java 回收器将某个指向对象 O 的软引用置空，它必须在同一时刻自动^②将所有导致对象 O 强可达的软引用置空。这一规则可以确保在回收器将这些引用置空之后，对象将不再软可达。

弱引用 (weak reference)：一旦回收器发现某一（软*可达的）弱引用的目标对象变为弱可达，则回收器必须将该引用置空（从而确保其目标对象不再软*可达）。与软引用类似，一旦回收器将某个指向对象 O 的弱引用置空，则必须同时将所有其他导致该对象软*可达的软*可达弱引用置空。

终结方法引用 (finaliser reference)：我们将从终结表到待终结对象的引用称为终结方法引用。我们曾在第 12.1 节描述了 Java 的终结机制，此处再次进行描述是为了说明此类引用的相对强度。终结方法引用只在运行时系统内部出现，它并不会像弱对象一样暴露给开发者。

223

虚引用 (phantom reference)：Java 中最弱的一种引用类型。虚引用只有与通知机制联合使用才具有一定意义，这是因为虚引用对象不允许程序经由该引用获取目标对象的引用，因此程序唯一可能的操作是将虚引用置空。程序必须显式地将虚引用置空来确保回收器将其目标对象回收。

Java 语言中不同强度的引用并没有我们此处描述的这么多，但此处的每种语义却与语言规范所定义的每种弱引用相关联。软引用允许系统对可调整的缓存进行收缩；弱引用可以用于规范化表或者其他场景；虚引用允许开发者控制回收的顺序以及时间。

多强度引用的实现需要在回收过程中增加额外的遍历，但这些操作通常可以很快完成。下面我们以 Java 的 4 种强度为例来描述复制式回收器对不同强度引用的处理方式。标记—清扫回收器也可以通过类似的方式实现，同时也会更加简单。引用计数回收器中的实现方式我们将稍后介绍。回收器应当以如下方式执行堆遍历过程：

1) 从根开始，仅对强可达对象进行追踪并复制，同时找出所有的软对象、弱对象、虚对象（但不对这些对象进行追踪）。

2) 如果必要，则自动将所有软引用置空^③。如果无需将软引用置空，则需对其进行追踪

① 当前为止，除了 Java 之外，我们尚未发现其他语言支持多种强度的弱引用，但这一思想可能会在未来得到广泛借鉴。

② 自动这一概念在 Java 规范中的含义是，没有任何线程会看到仅有部分引用被置空的状态，即要么所有软引用都未置空，要么全都置空。为达到这一目的，可以使用一个全局共享旗标来表示是否应当将（包含弱引用的）对象的引用域看作已置空，即使在该引用域尚未被置空的情况下。对象自身内部需要包含一个旗标来表示全局共享旗标是否应当生效，即引用是否应当被看作是已置空。在并发回收器中，使这一操作能安全执行可能需要引入适当的同步操作。

③ 也可以选择性地置空软引用，但这在实现上具有一定的难度。这里的“全部”是指所有已经存在的软引用，而不仅仅是回收器刚刚发现的。

和复制,并找出所有的软*可达对象。对软可达对象进行追踪时可能会发现新的弱可达或虚可达对象。

3) 如果弱引用的目标对象已被复制到目标空间,则更新该引用,否则将该引用置空。

4) 如果尚未复制的对象中存在需要终结的对象,则将其加入终结队列,然后回到第 1 步,并以终结队列作为新的根进行追踪。需要注意的是,在第二轮执行过程中将不会再有需要终结的对象产生[⊖]。

5) 如果虚引用的目标对象并未得到复制,则将其加入 ReferenceQueue 里。然后回收器将从该对象开始完成虚*可达对象的追踪与复制。需要注意的是,回收器不会将任何虚引用置空,这一操作必须由开发者显式操作。

尽管上述步骤是以复制式回收器作为原型来描述的,但这一过程同样也适用于标记-清扫回收。但是,为 Java 的弱引用语义设计引用计数版本的实现却显得相当困难。一种实现策略是在正常的引用计数中忽略来自软引用、弱引用、虚引用的贡献(统称为非强引用),并在对象中使用一个独立的位来反映其是否存在非强引用来源。我们同样也可以通过一个独立的位来记录对象是否包含终结方法。除此之外,系统可能还需要通过一张全局的表来记录每个弱引用目标对象的引用来源。我们将该表称为反向引用表(Reverse Reference Table)。

224

由于引用计数并不会像追踪式回收器一样发起单独的回收调用,所以必须引入一些其他的启发式方法来判定何时置空软引用,而这一操作需要自动执行。在这一方案下,最简单的实现方案可能是将对象的软引用来源也(像正常引用一样)计入引用计数中,而当其强引用被置空时,回收器将使用启发式方法来判定是否需要回收该对象,以及是否需要处理其更弱的来源引用。

如果某一对象的正常引用(强引用)变为零,则该对象将被回收(同时减少其子节点的引用计数),除非其是非强引用的目标并需要终结。如果对象头部的标记位显示其存在至少一个非强引用来源,则我们必须从反向引用表中找出其所有非强引用来源,并执行如下所示的流程来处理这些引用对象(Reference 对象)。引用的处理应当依照从强到弱的顺序。

弱引用:将弱引用置空,并将需要终结的对象添加到待终结队列。

终结方法引用:将对象加入待终结队列,这一操作需要正常增加对象的引用计数,因此其引用计数将会恢复到 1。与此同时,清空对象头部中包含有终结方法的位。

虚引用:如果目标对象存在终结方法,则无需任何操作。如果该对象存在来自虚对象的引用,则需将虚对象加入用户指定的队列中,同时标记其已经入队。为避免虚引用的目标对象被回收,需要增加目标对象的引用计数。当用户显式置空虚对象中的虚引用时,如果其已经入队,则减少目标对象的正常引用计数值,回收器在自动回收虚对象时也需如此。

除此之外有一些特殊的情况需要注意。当回收某一 Reference 对象时,我们需要将其从反向引用表中移除,同理,当用户显式清空 Reference 对象时也需执行相同的操作。

当回收器发现环状垃圾时还有一些额外的技巧。在对环状垃圾进行处理之前,我们需要先判断其中的某个对象是否存在软引用来源,若结果为真,则将环状垃圾全部保留,但仍需周期性地这一检测。如果环状垃圾中的所有对象都不存在软引用来源,但某些对象存在

⊖ Barry Hayes 指出,在如下场景可能会出现:假设弱对象 w1 从某个需要终结的对象 x 可达,同时 w1 又引用了另一个需要终结的对象 y,而 y 又被另一个弱对象 w2 引用。也就是说,w1 和 w2 两个弱对象均引用了对象 y。如果 w2 强可达,则其将被置空,但如果 w1 仅从 x 可达,则其将会保留。这一情况下,如果对象 y 的终结方法将其复活则会产生问题,因为对象 y 的历史不可达导致 w2 被置空,但此时对象 y 又重新变成强可达。不幸的是,这一问题似乎是 Java 所定义的弱对象与终结化机制与生俱来的问题。

弱引用来源（即存在来自弱对象的引用），则需要自动将这些弱对象全部置空，并将所有需要终结的对象加入待终结队列。最后，如果上述情况都未发生，但环中的某些对象存在虚引用来源，则需要保留整个环，并将引用环中对象的虚对象加入用户指定的队列中。如果环中的所有对象都不存在非强引用来源，且都不需要终结，则可以回收整个环。

12.2.3 使用虚对象控制终结顺序

假设有两个对象，A 和 B，我们希望它们的终结顺序是先 A 后 B。一种实现策略是创建虚对象 A' 来持有对象 A 的虚引用。除此之外，A' 的类型应该是对 Java 的 PhantomReference 的扩展，其将持有一个指向对象 B 的强引用，以避免对象 B 被提早终结[⊖]。图 12.5 演示了这一情况。

一旦对象 A'（即对象 A 的虚引用来源）被加入到用户指定的队列，意味着对象 A 已经从应用程序不可达，并且 A 对应的终结方法已经运行过，这是因为终结队列可达要比虚可达的强度更高。然后将虚对象 A' 指向 A 的虚引用置空，再将其指向 B 的强引用置空，如此一来，对象 B 的终结方法将在下一轮回收中得到调用。最后我们再把虚对象从全局对象表中删除，则虚对象本身也将得到回收。我们很容易将该策略进行推广，进而实现三个或者更多对象的终结顺序控制，所付出的代价是在两两对象之间通过虚对象来施加终结顺序限制。

终结顺序的控制只能通过虚对象完成，弱对象无法胜任。对于图 12.5 所示的状态，如果我们将虚对象替换为弱对象，则当对象 A 不可达时，A' 中的弱引用将被置空，且 A' 将被添加到用户指定的队列中。此时我们可以将 A' 中指向 B 的强引用置空，但不幸的是，置空 A' 中弱引用的操作将发生在 A 的终结方法执行之前（因为弱引用的强度高于终结方法引用——译者注），此时我们将很难知道 A 的终结方法何时执行完毕，因此对象 B 的终结方法可能会先得到执行。故意将虚引用的强度设计得比终结方法引用更低，目的正在于确保只有当对象的终结方法执行完毕之后，其虚引用来源才会被加入到用户指定的队列。

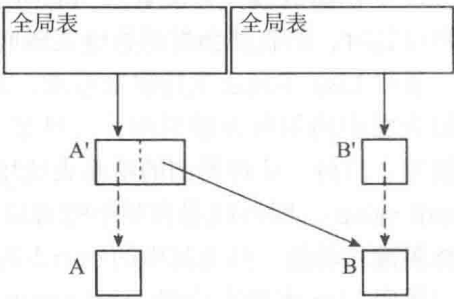


图 12.5 按序终结。我们希望在对象 A 和 B 同时不可达时相应的终结顺序为先 A 后 B。虚对象 A' 包含了对对象 A 的虚引用以及对象 B 的强引用

225

12.2.4 弱指针置空过程的竞争问题

在 12.1 节我们提到，某些特定的编译优化可能会引发竞争，进而可能导致终结方法被过早地调用。弱指针也存在同样的问题，这一竞争也可能导致弱指针被过早地置空。

12.2.5 弱指针置空时的通知

在弱引用机制之上，某些应用程序可能需要在特定的弱引用被清空时得到通知（虚引用可以通知应用程序对象已被终结，而弱引用则可以通知应用程序对象可能将要终结），然后再执行某些适当的动作。因此，弱引用机制通常也会支持通知，其实现策略一般是将弱对象添加到开发者指定的队列中。例如，Java 的 ReferenceQueue 内建类便是以此为目的设计的，应用程序既可以对其进行轮询，也可使用阻塞式的操作来获取元素（或者附加额外的超时时间）。

⊖ 在 Java 内建引用类的派生类中增加一个域来持有强引用（而非新增一种特殊的弱引用）。

间)。应用程序也可以检测某个给定的弱对象是否已经加入到某个队列中 (Java 只允许弱对象最多被加入到一个队列中)。回收器在对弱指针的多次遍历过程中可以很轻松地实现弱对象的入队。许多语言都增加了类似的通知机制。

12.2.6 其他语言中的弱指针

226

Java 支持多种强度的弱引用, 因而我们单独对其进行描述。除此之外, 其他语言还支持一些不同的或者额外的弱引用特征。

许多 Lisp 实现支持弱数组与弱向量[Ⓐ], 它们仅仅是弱对象的简单集合: 当某一对象 O 不再强可达时, 回收器会将弱数组或弱向量中所有指向 O 的弱引用置空。

某些 Lisp 实现还支持弱哈希表, 其通常包括 3 种类型。第一种类型为弱键 (weak key), 即键为弱引用但值为强引用, 一旦某个键-值对的键不再强可达, 回收器将从哈希表中移除该键-值对。此种类型的哈希表适用于规范化表或者特定缓存的实现。第二种类型为弱值 (weak value), 即回收器在某个键对应的值不再强可达时移除该键-值对。第三种类型同时支持弱键和弱值, 只要其中的一个不再强可达, 回收器便会移除该键-值对。

某些 Lisp 实现支持弱-与 (weak-AND) 和弱-或 (weak-OR) 对象, 这些对象都属于潜在的弱对象, 其工作模式如下。只要弱-与对象有一个元素不再强可达, 回收器便会将其中的所有引用置空, 这与 Lisp 中的 AND 运算符类似, 即只要有一个参数为空, 则返回值为空。与之相对的是弱-或对象, 只有当其中的所有元素都不再强可达之后, 回收器才会将其中的所有引用置空。读者可以从 Platform Independent Extensions to Common Lisp[Ⓑ] 获取更多的细节以及结论, 包括弱关联 (weak associations)、弱与-映射 (weak AND-mapping)、弱或-映射 (weak OR-mapping)、弱关联列表 (weak association list), 以及我们前面曾经讨论过的弱哈希表。

寄生对组 (ephemerons) [Hayers, 1997][Ⓒ] 是弱键-值对组的一种特殊实现形式, 它可以用于维护依附于其他对象的信息。假设我们需要通过额外的表将信息 I 与对象 O 相关联, 此时我们便可使用寄生对组, 并以 O 作为键、以 I 作为值。寄生对组的语义如下: 寄生对组对键的引用为弱引用, 当键不为空时, 其对值的引用为强引用, 反之, 当键变为空时, 其对值的引用将变为弱引用。在我们的案例中, 初始情况下寄生对组对 O 的引用为弱引用, 对 I 的引用为强引用。一旦 O 被回收, 则键的弱引用将被置空, 这将导致其对 I 的引用降级为弱引用。因此, 只要对象 O 依然存活, 信息 I 便不会被回收, 而一旦对象 O 死亡, 则信息 I 便可能成为垃圾并被回收。具有通知机制的弱键/强值对组或多或少可以模拟寄生对组的实现 (收到键被置空的通知之后将值置空, 或者将值降级为弱引用), 但一个细微的区别在于, 即使寄生对组中的键从其值强可达, 也不会对键的可达性造成影响。也就是说, 即使 I 引用了 O, 寄生对组依然可以确保 O 能够得到回收, 但弱对组实现方式则不可能将 O 回收。如果不借助于寄生对组, 则达到相同目的的唯一方式是确保从 I 到 O 的任意一条路径中都至少包含一个弱引用。

寄生对组的实现概要如下 (我们忽略弱指针或终结机制的实现形式)。首先, 从根开始对强指针进行追踪, 如果发现寄生对组, 则只记录而不追踪。然后对寄生对组集合重复进行

Ⓐ 数组可能会是多维的, 而向量则通常只是一维的, 它们之间的区别并不影响弱引用语义。

Ⓑ <http://clisp.cons.org>.

Ⓒ Hayers 把寄生对组的发明归功于 George Bosworth。

如下迭代：如果某一寄生对组的键强可达，则从其值开始进行追踪，并将其从寄生对组记录中移除。这一过程可能会追踪到其他寄生对组的键，也可能会发现新的寄生对组并将其记录。最终我们将得到一个集合，该集合要么为空，要么其中寄生对组的键均不可达。最后，我们将这些寄生对组的键置空，如果其值不可达也将其置空。另外，我们还可以使用通知机制并将键不可达的寄生对组加入到某一队列，但这可能会引入对象的复活问题。或许更好的解决方案是清空寄生对组的域，并将其键值组成一个新的普通对组传递给终结函数。

[227]

在 Java 和 Lisp 之外，许多其他语言（或其某些实现）也支持弱引用，包括 Action Script、C++（例如 Boost 库）、Haskell、JavaScript、OCAML、python、Smalltalk 等。还有一些语言已经计划在其未来的版本中引入弱引用语义 [Donnelly 等，2006]。

12.3 需要考虑的问题

本章我们介绍了终结机制以及弱指针这两个“特定语言相关”的概念，它们基本已被认为是自动内存管理领域的一部分。自动内存管理对于软件工程学来说十分重要，以此为基础可以更简单地实现复杂系统的正确构建，但在此之外，为解决一些特定的问题又衍生出终结机制和弱指针等编程语言扩展语义——而自动内存管理机制则是这些扩展语义的实现基石。

如果需要为指定的语言设计回收器和运行时系统，则设计者所需关注的大部分内容都已经确定，即语言自身的要求是什么。本章我们所涉及的内容，特别是在终结方面所要做出的种种选择，更多是在新语言的设计中需要考虑的问题。类似地，弱指针机制也存在多种选择，例如应当提供哪些“弱”数据结构，需要提供多少种强度的引用等，这些同样也是设计新语言时需要多考虑的问题。

如果编程语言需要支持终结机制与弱引用，那么回收器和运行时系统的实现者需要更加关注分配与回收技术的选择与设计，包括：

- 弱指针与终结机制通常需要额外的追踪过程。这些额外的追踪过程通常可以很快完成，因此通常不会造成性能问题。但是，这些机制的引入会增加基础回收算法的复杂性，进而需要进行相当仔细的设计。因此，最好是在设计之初便考虑这些因素，而非在后期引入这些机制。毋庸置疑，对终结机制和弱引用这两种语义的深入理解，对其推荐实现策略的真正掌握，对于编程语言的实现都是十分重要的。
- 某些机制（特别是 Java 中某些强度的弱引用）要求回收器在同一时刻将堆中所有相关弱引用全部置空。这一要求在万物静止式回收器中实现起来相对容易，但在并发环境下却需要一些额外的机制来保证。对弱引用进行遍历可能需要对某一共享标记进行检查，还可能引入额外的同步机制来保证回收器和赋值器线程对存在弱引用的对象做出相同的存活性判定，也就是说，当回收器尝试原子性地清除一组弱引用时，赋值器可能会在同一时刻尝试获取某个相关对象的强引用，这一过程中可能存在的竞争问题需要引入同步来解决。这一竞争问题并非 Java 弱引用机制的特有问题，而是所有同时支持弱指针和并发回收的语言都会面临的潜在问题。
- 弱指针和终结机制的实现复杂度较高，且在某些情况下我们还需要原子性地将一组弱引用集置空，因此我们可以将其处理过程放在并发回收器的万物静止阶段，或者使用锁来替代更加复杂的无锁（lock-free）或者无等待（wait-free）技术。我们将在第 13 章介绍如何控制对数据结构的并发访问。
- 在 Java 语言中，回收器还需要在每个回收周期中判定是否需要将软引用置空。

[228]

并发算法预备知识

并发回收算法早在 20 世纪 70 年代就开始得到研究 [Steele, 1975]。尽管长期以来多处理器技术仅应用在很少一部分场景下，但在当下，多处理器技术已经得到了大范围的商业化应用，甚至连用于本书写作的笔记本电脑都拥有四核处理器。除此之外，处理器时钟速度的增长已经不再像过去几十年那样可以带来性能的飞跃，因而开发者提升程序速度的唯一途径只能是充分利用多核来处理相同的任务。因此，编程语言实现需要支持并发编程，其运行时系统和垃圾回收器也应当对并发环境有着良好的支持。后续章节将深入介绍并行回收 (parallel collection)、并发回收 (concurrent collection)、实时回收 (real-time collection)，但在此之前，我们首先需要了解这些回收技术在逻辑和物理并行方面所依赖的基本概念、算法、数据结构，包括：硬件相关内容、内存一致性模型 (memory consistency model)、原子更新原语、前进保障 (progress guarantee)、互斥算法、工作共享与结束检测 (termination detection)、并发数据结构，最后将介绍一种新兴模型：事务内存 (transactional memory)。

13.1 硬件

为了理解并行回收与并发回收的正确性以及性能，我们有必要先掌握多处理器硬件的相关特性。本节给出了如下几个关键概念的定义和综述，包括：处理器与线程（多处理器、多核、多程序、多线程）、互联 (interconnect)、内存与高速缓存[⊖]。

13.1.1 处理器与线程

处理器 (processor) 是硬件执行指令的单元。线程 (thread) 是单一顺序控制流，是软件执行的具体化。线程的状态可以是运行中 (running) (也称调度中 (scheduled))、可运行 (ready to run)，或者是为等待某些条件而被阻塞 (blocked)，例如等待消息的到来、输入/输出的完成，或者到达特定的时间。调度器 (scheduler) 通常是操作系统组件，其功能是确定在任意时刻哪些线程应当在哪些处理器上执行。如果某个线程被调度器从某个处理器换出 (其状态从运行中转变为可运行或者被阻塞)，则当其下一次被调度时很可能在另一个不同的处理器上运行。当然，调度器也允许线程和处理器之间存在一定的亲和性 (affinity)。

[229]

某些处理器硬件支持多个逻辑处理器共用一条指令流水线，该技术称为同时多线程 (simultaneous multithreading, SMT) 或者超线程 (hyperthreading)。这一概念会给我们的定义带来一定的麻烦。在我们的术语中，逻辑处理器通常被称为线程，但在此处，同时多线程处理器则可以看作是多 (逻辑) 处理器，并可以独立进行调度，因而线程这一概念便只能代表软件实体。

多处理器 (multiprocessor) 是包含多个处理器的计算机。片上多处理器 (chip multiprocessor, CMP) 是指在单个集成电路芯片上集成多个处理器的技术，也称多核处理器 (multicore processor) 甚至众核处理器 (many-core processor)。抛开并发多处理器的概念不谈，多线程 (multithread) 是

⊖ 本节的相关内容从 Herlihy 和 Shavit [2008] 的文献中受惠颇多，读者可以通过该书获取更多的相关内容。

指使用多个线程的软件，且每个线程可能在多个处理器上并发运行。多程序（multiprogram）是指在单一处理器上执行多个进程或者线程的软件。

13.1.2 处理器与内存之间的互联

多处理器与集群计算、云计算或者分布式计算的区别在于，前者存在每个处理器都可以直接访问的共享内存。处理器对共享内存的访问需要以某种互联网络作为媒介。最简单的互联方式是处理器和内存之间使用单个共享总线（bus）来传递信息。我们可以简单地将内存访问操作看作是处理器和内存单元之间的消息通信，每次通信所需的时间可能会达到上百个处理器时钟周期。单个总线的原始速度相当快，但该速度在多处理器同时发起请求时却仍然会成为瓶颈。带宽最高的互联方式可能是在处理器和内存两两之间建立私有通道，但该方案所需的硬件资源却会正比于处理器和内存单元数量的乘积。为获取更高的整体带宽（整个系统中处理器和内存在一秒内可以传输的数据量），将内存分割成多个单元也是一种不错的方案。另外，处理器与内存之间的数据传输通常都是以高速缓存行（参见 13.1.4 节）而非单独的字节或者字节为单位的。

对于更大的片上多处理器，一次内存访问请求在互联网络中的传递可能需要经过多个节点，例如当互联网络以网状或者环状方式组织时。此处的具体细节超出本书的讨论范围，但我们需要了解的是，内存的访问时间可能会随着处理器与内存单元在互联网络中的位置不同而发生变化。另外，相同互联路径上的并发访问也可能引发更大的延迟。

在单总线系统中，当处理器的数量达到 8 ~ 16 个之后，总线一般都会成为瓶颈。但相比其他互联方式，总线的实现通常更加简单且更加廉价，且总线允许每个单元侦听（listen）总线中的所有通信（有时也称为窥探（snooping）），这可以简化系统对高速缓存一致性的支持（参见 13.1.5 节）。

如果内存单元与处理器之间相互独立，则该系统可以称为对称多处理器（symmetric multiprocessor, SMP）架构，该架构中每个处理器访问任意内存单元的时间都是相同的。我们同样也可以将内存与每个处理器相关联，此时处理器在访问与自身关联的内存时速度更快，而在访问与其他处理器关联的内存时则速度较慢。此类系统被称为非一致内存访问（non-uniform memory access, NUMA）架构。同一系统可以同时包含全局的 SMP 内存以及 NUMA 内存，每个处理器还可以拥有私有内存，但共享内存与垃圾回收技术的关联更大[⊖]。

230

对于处理器与内存之间的互联，最值得注意的地方是内存访问需要花费较长的时间，且互联网络可能成为系统的瓶颈，与此同时，不同处理器访问内存的不同部分可能会花费不同的时间。

13.1.3 内存

尽管内存存在物理上可能会跨越多个内存单元或者处理器，但从垃圾回收器的角度来看，共享内存看起来就是一块由字或者字节组成的单个地址空间。由于内存是由多个可以并发访问的单元组成的，所以我们无法对其在任意时刻的状态给出一个全局性的描述，但是内存中的每个单元（也就是每个字）在每个时刻的状态都是确定的。

⊖ 如果可以将线程绑定到特定处理器，则处理器私有内存可以用于线程本地堆的实现（但是，如果某一线程的本地堆驻留在特定处理器私有内存中，该线程将只能在该处理器上执行）。处理器私有内存也可用于存放不可变数据的本地副本。

13.1.4 高速缓存

由于内存的访问速度如此之慢，所以现代体系架构通常会在处理器和内存之间增加一到多层高速缓存，其中所记录的是处理器最近访问过的数据，进而降低了程序运行期间处理器需要访问内存的次数。高速缓存与内存的数据交换是以高速缓存行（也称高速缓存块）为单位的，通常为 32 或者 64 字节。如果处理器在访问某一地址时发现其所需要的数据已经存在于高速缓存中，这一情况称为高速缓存命中（cache hit），反之则称高速缓存不命中（cache miss），此时处理器便需访问更高一级缓存，如果最高一级缓存依然不命中，则处理器必须访问内存。片上多处理器（CMP）中某些处理器可能会共享最高一级缓存，例如，每个处理器可能都拥有专属的 L1 高速缓存，但是其会与相邻的一个存储器共享 L2 高速缓存。各级高速缓存的缓存行大小可以不同。

当某一级缓存出现不命中，且该级缓存也无法容纳新的缓存行时，处理器就必须依照某种策略从中选择一个缓存行进行置换，被换出的缓存行称作受害者（victim）。当在缓存中写入数据时，某些缓存使用的是写通（write-through）策略，即当某一缓存行中的数据得到更新时，下一级缓存中的对应数据也会尽快得到更新。另一种策略是写回（write-back），即在被修改的行（也称脏行）得到换出之前其中的数据不会写入下一级缓存，除非进行显式刷新（flush）（需要使用特殊的指令）或者显式写回（也需要特殊指令支持）。

缓存置换策略在很大程度上依赖于缓存的内部组织形式。全相联（fully-associative）缓存允许内存中任意地址的数据放置到缓存的任意一行中，其置换策略也可选择任意一行进行淘汰。与之对应的另一个极端是直接映射（direct-mapped）缓存，即内存中某一地址的数据只能放置到缓存中特定的行，因而其置换策略只可能淘汰特定的缓存行。 k 路组相联（ k -way set-associative）缓存是上述两种极端方案的折中，该策略允许内存中特定地址的数据映射到缓存中的 k 个缓存行，其置换策略也可从这 k 个缓存行中选择一个进行淘汰。这三种基本的缓存置换策略还存在其他一些变种，例如受害者缓存（victim cache），该缓存包括一个使用直接映射策略的主缓存以及一个额外的容量较小的全相联缓存，从主缓存中淘汰的行将被置于全相联缓存中。该策略的缓存相关性较高，且硬件开销较小。

高速缓存设计中需要注意的另一方面是各级缓存之间的关系。对于相邻两级缓存，如果较低级别缓存中的数据一定会在高级别缓存中存在，则两级缓存为（严格）包容（inclusive）关系。相反，如果同一数据最多只能出现在两级缓存中的一级，则两级缓存为排他（exclusive）关系。真正的高速缓存设计也可进行折中，即允许同一行存在于两级缓存中，但也并不强制要求高级别缓存一定要包容低级别缓存的数据。

231

13.1.5 高速缓存一致性

高速缓存中所持有的数据在内存中很可能是共享的。由于每个缓存中的数据不可能同时得到更新（特别是对于使用写回策略的缓存），所以内存中同一地址的数据在不同缓存中的副本可能会出现不一致。因此，不同处理器在同一时刻读取同一地址的数据，可能会获得不同的结果，这显然是不应该出现的。为解决这一问题，底层硬件通常会提供一定级别的高速缓存一致性支持。一种经典的高速缓存一致性协议（coherence protocol）是 MESI，在该协议中，每个缓存行可能有 4 种状态，这 4 种状态的首字母构成了该协议的名称。

被修改（modified）：该缓存行持有数据的唯一有效副本，其中的数据被修改过，但尚未写回内存。

独占 (exclusive): 该缓存行持有数据的唯一有效副本, 同时其中的数据与内存保持一致。

共享 (shared): 其他缓存行也可能持有数据的有效副本, 同时所有副本中的数据均与内存保持一致。

无效 (Invalid): 缓存行中不包含任何有效数据。

只有当缓存行的状态为“被修改”、“独占”、“共享”其中之一时, 处理器才可以读取该缓存行; 只有当缓存行的状态为“被修改”或“独占”时, 处理器才可以将数据写入该缓存行, 写入之后其状态将成为“被修改”。如果处理器需要从“无效”缓存行中读取数据, 则系统的后续行为取决于该缓存行在其他缓存中的状态: 如果为“被修改”, 则处理器必须将其写回内存, 并将其状态置为“共享”(或者“无效”); 如果状态为“独占”, 则只需将其降级为“共享”(或“无效”); 如果其状态为“共享”或者“无效”, 则处理器只需简单地从内存或者其他缓存里状态为“共享”的缓存行中加载数据。如果处理器需要将数据写入“无效”缓存行中, 系统的后续行为与读取时的行为类似, 唯一的不同之处在于其他缓存行的最终状态都将是“无效”。如果处理器需要将数据写入“共享”缓存行中, 其必须先将其其他缓存行降级为“无效”。该协议可以进行的改进包括: 以写为目的读可以在读取完成之后将其其他缓存行的状态降级为“无效”; 写回操作可以将缓存行的状态从“被修改”降级为“独占”; 令某一缓存行失效的操作可以将状态为“被修改”的缓存行写回内存, 然后将其状态置为“无效”。

MESI 协议的关键之处在于, 任意缓存行在同一时刻只能被一个处理器写, 且两个缓存针对同一数据的缓存行永远不会产生不一致。MESI 协议的实现难点在于, 当处理器数量增大时算法的性能会下降, 这也是所有由硬件支持的缓存一致性协议的共有问题。因此, 更大的片上多处理器逐渐开始放弃内建的缓存一致性协议, 转而开始由软件来管理一致性, 此时软件便可选择任意类型的缓存一致性协议。即便如此, 处理器数量增大时算法依然会存在性能问题, 但与将算法固化在硬件中的策略相比, 开发者至少可以根据其具体需求选择更好的一致性算法。

缓存一致性要求引发了另一个问题, 即伪共享 (false sharing): 当两个处理器同时访问并更新位于相同缓存行的不同数据时, 由于处理器在写操作之前必须将缓存的状态更改为“独占”, 所以两个处理器令对方缓存行失效的操作会产生“乒乓”效应, 从而引发互连网络中大量的一致性通信, 并可能引发额外的内存读取操作。

13.1.6 高速缓存一致性对性能的影响示例: 自旋锁

典型的互斥锁可以通过 `AtomicExchange` 原语实现, 如算法 13.1 所示。后续描述中我们均以首字母大写的方式来区分原子指令原语, 我们同时也使用首字母小写的 `load` 和 `store` 来表示低级读写操作, 并以此避免与应用程序和赋值器之间的读写接口混淆。锁的初始值应该为零, 意味着未上锁。如果尝试加锁失败, 处理器将在 `while` 循环中自旋, 因而称其为自旋锁。每次循环中, 原子化的“读-修改-写”操作都会尝试独占其所在的高速缓存行, 因而如果多处理器竞争该锁, 高速缓存行将会出现“乒乓”效应, 即使对于已经持有锁的处理器也不例外。更加糟糕的是, 即使持有锁的处理器想要释放锁, 其也需要与其他处理器竞争该高速缓存行的独占权。此种自旋锁实现方式被称为“检测并设置”锁 (`test-and-set lock`), 尽管它并未依赖我们后文将要介绍的 `TestAndSet` 原语。

算法 13.1 基于 AtomicExchange 原语的自旋锁

```

1  exchangeLock(x):
2      while AtomicExchange(x, 1) = 1
3          /* 空 */
4
5  exchangeUnlock(x):
6      *x ← 0
7
8  AtomicExchange(x, v):
9      atomic
10         old ← *x
11         *x ← v
12     return old

```

算法 13.1 所描述的自旋锁的实现方式会导致最严重的一种竞争情况出现，因而算法 13.2 所描述的“检测－检测并设置”锁（test-and-test-and-set lock）作为一种更加巧妙的改进版本，在许多程序中得到应用。其最大的改进之处在于第 9 行，算法在调用 AtomicExchange 方法之前先通过一般的读操作判断锁是否已被占用，因而此处的自旋操作只需访问处理器自身的（已经保持一致）的高速缓存，无需进一步访问总线。如果锁位于不可缓存（noncacheable）的内存中，则该线程可以使用空循环来等待，也可在检测之间插入硬件 idle 指令，如果等待时间稍长，则在两次检测之间的等待时间可以呈指数级别增加或者采用其他类似算法。如果等待时间过长，则线程可以请求操作系统的调度器介入并放弃剩余时间片，或者转而采取等待某一显式信号的方式，此时便要求锁的持有者在释放锁时发送信号。

算法 13.2 基于 AtomicExchange 原语的“检测－检测并设置”自旋锁

```

1  testAndTestAndSetExchangeLock(x):
2      while testAndExchange(x) = 1
3          /* 空 */
4
5  testAndTestAndSetExchangeUnlock(x):
6      *x ← 0
7
8  testAndExchange(x):
9      while *x = 1
10         /* 空 */
11     return AtomicExchange(x, 1)

```

233

13.3 节涵盖了大部分常用的硬件原子操作原语，因而在此我们有必要启发式地介绍如何以 TestAndSet 原语来实现“检测并设置”以及“检测－检测并设置”锁，如算法 13.3 所示。TestAndSet 原语的优势在于，锁的值与其语义上的含义隐式地保持一致，即 0 代表未加锁，1 代表已加锁。如果锁的值为 1，则处理器不必访问总线，也无需以独占方式访问高速缓存行。原则上讲，AtomicExchange 原语也具有相同的优点，但其需要判断锁的新老值是否相同，而不能简单地判断其是否为 1。

算法 13.3 基于 TestAndSet 原语的自旋锁

```

1  testAndSetLock(x):
2      while TestAndSet(x) = 1
3          /* 空 */
4

```



```

5 testAndSetUnlock(x):
6     *x ← 0
7
8 TestAndSet(x):
9     atomic
10         old ← *x
11         if old = 0
12             *x ← 1
13             return 0
14         return 1
15
16 testAndTestAndSetLock(x):
17     while testAndTestAndSet(x) = 1
18         /* 空 */
19
20 testAndTestAndSet(x):
21     while *x = 1
22         /* 空 */
23     return TestAndSet(x)
24
25 testAndTestAndSetUnlock(x)
26     testAndSetUnlock(x)

```

13.2 硬件内存一致性

我们假定共享内存可以提供与高速缓存相同的一致性 (coherence) 保障, 即: 不存在未完成的写操作。如果两个处理器读取内存中相同位置的值, 所获取的值也是相同的。大多数硬件还可以进一步保证: 如果两个处理器同时对内存的相同位置发起写操作, 则其中的一个将先于另一个发生, 同时所有处理器后续读取到的值都将是最后一次写入的值。另外, 如果某一处理器已经读取到了最终的值, 则在下一次写操作发生之前, 其不可能读取到其他值[⊖]。也就是说, 针对内存中任何特定位置的写操作都是经过排序的, 且在任意处理器看来, 特定位置值的变化顺序都是相同的。

[234]

但是, 对于程序在多个位置的写入 (或者读取) 操作, 硬件系统却不能保证程序发起操作的顺序与其在高速缓存或者内存中的生效顺序完全一致, 更不能保证其他处理器能够以相同的顺序感知到这些地址的数据变更。也就是说, 程序顺序 (program order) 不一定要与内存顺序 (memory order) 完全一致。读者可能不禁会问: 为何如此? 其中的含义是什么? 前一个问题关乎于硬件和软件, 概括来讲, 不要求两者完全一致是出于性能考虑——严格一致性 (consistency)[⊖] 要么会耗费更多的硬件资源, 要么会降低性能, 或者两者兼有。对于硬件而言, 许多处理器会使用写缓冲区 (write buffer/store buffer) 来保存未完成的内存写入操作。写缓冲区本质上是一个 <地址, 数据> 对组队列。正常情况下, 写操作可能会有序执行, 但如果某一后发写操作的目的地址已经存在于写缓冲区中, 则硬件可能会将其合并到队列里尚未完成的写操作中, 这便意味着写操作可能会出现“后发先至”的情况, 即较晚的写

⊖ Java 的内存模型更加宽松: 如果两个写操作之间未进行同步, 则允许处理器在后续执行过程中读取到任意一次写入的值, 此时该值将可能出现“振荡”。

⊖ coherence 和 consistency 的中文表达均为“一致性”, 但两者存在区别: coherence 主要考虑多处理器对同一内存位置的写操作对于所有的处理器的可见性; 而 consistency 则主要考虑程序发起操作的顺序与其在内存中的生效顺序之间的关系, 且其主要是针对不同地址。——译者注

操作可能会越过较早的、针对其他地址的写操作而立即在内存中生效。处理器的设计者会小心地确保处理器操作针对其自身的一致性，也就是说，如果处理器在读取某一位置的值时发现该位置存在未完成的写操作，则处理器要么通过直接硬件路径（速度较快但开销较大）来读取该值，要么必须等写缓冲区刷新完毕后再从高速缓存中读取该值。另一个可能导致程序操作被重排序的原因是高速缓存不命中。一旦读取过程中发生高速缓存不命中，许多处理器会将其跳过并继续执行后续指令，进而可能出现后发读 / 写操作越过先发读 / 写操作先执行完毕的情况。另外，对于使用写回机制的高速缓存，其中的数据只有在被淘汰或者显式刷新时才会写入内存，因此针对不同缓存行的写操作的执行顺序可能会出现大幅度调整。上述各种硬件方面的原因只是说明性的，但并非面面俱到。

由软件产生的重排序大多是由编译器造成的。例如，如果编译器已知两个引用指向的是同一地址，且两个引用的读取操作之间并无其他写操作会影响该值，则编译器可能直接使用其第一次读取到的值优化掉第二次读操作。更一般的情况是，如果编译器可以确保各变量之间均不存在别名关系（即不引用相同的内存地址），则其可以对这些变量的读写操作以任意方式重排序，因为不论采用何种顺序，最终的执行结果都是相同的（对于单处理器而言，且假定不存在线程切换）。读取结果复用以及重排序策略可以产生更高效的代码，且在大多数情况下并不影响语义，因而许多编程语言都允许这一策略。

从开发者的角度来看，程序顺序与内存顺序之间缺乏一致性显然是一个潜在的问题，但是从硬件实现者的角度来看，如此设计可以大幅提升性能并减少开销。

放宽一致性要求将会产生怎样的后果？第一，这可能导致程序的执行完全背离开发者的意图，也可能导致在完全一致性模型下可正常执行的代码在更加复杂的一致性模型下产生混乱的执行结果。第二，锁相关等技术要求硬件以某种方式确保对不同地址的访问能够有序执行。各种顺序模型必须能够区别出 3 种主要的访问原语：读（read）、写（write）、原子（atomic）操作[⊖]。原子操作需要原子化的“读-修改-写”原语，该原语通常是条件性的，例如 TestAndSet。内存一致性对于依赖加载（dependent load）也十分重要，所谓依赖加载，即程序需要先从地址 x 加载数据，然后再从地址 y 加载数据，但第二次加载操作的地址 y 取决于地址 x 的加载结果。依赖加载的一个典型案例是沿着指针链进行追踪。在完全一致性之外，还存在着许多较弱的内存访问顺序模型，我们将选择其中较为常见的进行介绍。

[235]

13.2.1 内存屏障与先于关系

内存屏障（memory fence）是一种处理器操作，它可以阻止处理器对某些内存访问进行重排序。特别地，它可以避免某些访问指令在屏障之前发送（issue），也可以避免某些访问指令被延迟到屏障之后发送，或者两者皆可。例如，完全读内存屏障可以确保屏障之前的所有读操作都能先于屏障之后的读操作执行。

先于关系（happens-before）这一概念则更加规范化，它是指内存访问操作在存储中所应遵从的发生顺序。完全读内存屏障相当于是每两个相邻读操作之间施加了先于关系。原子操作通常会为其内部的所有子操作施加完全内存屏障：所有较早的读、写、原子操作都必须先于较晚的读、写、原子操作发生。在先于关系之外还存在其他一些模型，例如获取-释放（acquire-release）语义。在该模型下，获取操作（acquire operation）（可以将其看作是获取锁）

⊖ 某些作者使用“同步”（synchronising）这一术语，而本书在此使用“原子”（atomic）这一术语。“同步”这一术语涵盖了这些操作的原子性及其对执行顺序的影响，因此“同步”与“原子”是两个严格不同的概念。

能够阻止较晚操作在该操作之前发生，但较早的读写操作则可以在获取操作之后发生；释放操作（release operation）与之完全对称：它能够阻止较早的操作在释放操作之后发生，但是较晚的操作则可以在释放操作之前发生。简而言之，处理器可以将获取－释放操作对之外的操作移动到其内部，但却不能将其内部的操作移动到外部。临界区（critical section）便可使用获取－释放模型来实现。

13.2.2 内存一致性模型

最强的内存一致性模型当属严格一致性（strict consistency），即所有的读、写、原子操作在整个系统中的任意位置都以相同的顺序发生（occur）[⊖]。严格一致性意味着所有操作的发生顺序都满足先于关系，且这一顺序是由某一全局时钟决定的。严格一致性是最容易理解的一种模型，这可能也是大多数开发者以为硬件系统所遵从的顺序，但这一模型很难高效地实现[⊖]。稍弱的一种模型是顺序一致性（sequential consistency）模型，在该模型中，全局先于顺序只需要与每个处理器的程序顺序保持一致即可。相对于其他更加宽松的一致性模型，顺序一致性模型下的编程更加简单，因而规模较小的处理器通常会尝试达到或者接近顺序一致性的要求。弱一致性（weak consistency）会将所有原子操作当作完全屏障。上文所描述的获取－释放模型通常被称为释放一致性（release consistency）模型。因果一致性（causal consistency）模型的强度介于顺序一致性和弱一致性之间，该模型要求程序所发起的读操作与其后续的写操作之间必须满足先于关系，其目的在于避免读操作对写操作所写入的值造成影响，即对于先读取某个值然后再将其写入内存的操作，因果一致性会确保它们之间的先于关系。宽松一致性（relaxed consistency）模型泛指所有比顺序一致性模型更弱的模型。

236

硬件系统所允许的重排列方式一定程度上还取决于互连网络和内存系统，这便超出了处理器的控制范围。表 13.1 展示了部分知名处理器家族所允许的指令重排列方式，所有的处理器至少都实现了弱一致性或者释放一致性。关于内存一致性模型的更多内容可以参见 Adve 和 Gharachorloo [1995, 1996]。

表 13.1 内存一致性模型与可能的重排列方式

重排列方式	Alpha	x86-64	Itanium	POWER	SPARC TSO	x86
读→读	Y		Y	Y		
读→写	Y		Y	Y		
写→写	Y		Y	Y		
写→读	Y	Y	Y	Y	Y	Y
原子操作→读	Y		Y	Y		
原子操作→写	Y		Y	Y		
依赖加载	Y					

注：Y 表示指定的操作可以不遵从先于关系，即可以进行重排列。

13.3 硬件原语

从某些最早的计算机开始，处理器就已经提供了原子化的“读－修改－写”原语来支持锁与同步。13.1 节介绍了两种原语：AtomicExchange 可能是最简单的一种硬件原语，该操

⊖ 此处的“发生”(occur)是指“看起来发生了”——程序无法区分这两者之间的区别。
⊖ 根据相对效应，完全有序在现代系统中甚至不存在一个明确的定义。

作既无需任何计算，也无需任何条件判断，它只是简单地将新值写入内存的某一地址，并原子性地返回该地址原有的值，且该操作不会与其他（原子性的或非原子性的）写操作发生交错；TestAndSet 原语同样也十分简单，它将单个位设置为 1，并返回该位以前的值，但该操作可以视为一种条件原语，因为只有当原始值为 0 时该操作才会将其设置为 1。其他比较知名且应用广泛的原子操作原语包括：“比较并交换”（compare-and-swap 或 compare-and-exchange），“加载链接 / 条件存储”（load-linked/store-conditionally 或 load-and-reserve/store-conditional）；各种不同的原子自增、原子自减原语，特别是“获取并增加”（fetch-and-add 或 exchange-and-add）。我们将顺次对其进行介绍。

13.3.1 比较并交换

237

算法 13.4 展示了 CompareAndSwap 原语以及与其十分相近的 CompareAndSet 原语。CompareAndSet 原语将某一内存地址的值与 old 比较，如果两者相等，则将其值置为 new。该操作的返回值表示内存中的值是否被更新。CompareAndSwap 原语与之的唯一区别在于其返回值是内存地址中原有的值（不论是否更新成功），而非一个布尔变量。尽管这两种原语的语义并非严格等价，但它们的使用场景基本都是一致的。

算法 13.4 CompareAndSwap 原语与 CompareAndSet 原语

```

1 CompareAndSwap(x, old, new):
2     atomic
3         curr ← *x
4         if curr = old
5             *x ← new
6         return curr
7
8 CompareAndSet(x, old, new):
9     atomic
10        curr ← *x
11        if curr = old
12            *x ← new
13        return true
14        return false

```

CompareAndSwap 通常用于将某一内存地址的值从一个状态更新到另一个状态，例如从“被线程 t1 锁定”到“解锁”，再到“被线程 t2 锁定”。算法 13.5 展示了 CompareAndSwap 原语的一种常见用法，即先判断内存地址的当前值，然后再尝试原子性地将其更新，该操作通常被称为“比较 - 比较并交换”（compare-then-compare-and-swap）。CompareAndSwap 原语潜藏着一个微妙的陷阱，即在调用 CompareAndSwap 时，目标内存地址的值改变了数次，但该地址的当前值却与调用者之前获取到的值相等。某些情况下这可能不会出现问题，但在其他情况下，相同的值并不意味着相同的状态。例如在垃圾回收中，在经历两次半区复制回收之后，某一指针的目标对象很可能与其最初的目标对象完全不同。CompareAndSwap 原语无法探测到“某个值被修改，然后再被改回原值”的情况，即所谓的 ABA 问题（ABA problem）。

算法 13.5 尝试使用“比较并交换”操作原子性地更新状态

```

1 compareThenCompareAndSwap(x):
2     if *x = interesting
3         z ← 表示下一状态的值
4         CompareAndSwap(x, interesting, z)

```

13.3.2 加载链接 / 条件存储

在 LoadLinked 和 StoreConditionally 原语中，处理器会记录 LoadLinked 原语所访问的地址，并使用处理器的一致性机制来探测所有针对该地址的更新操作，从而解决 ABA 问题。算法 13.6 描述了 LoadLinked/StoreConditionally 的具体实现，它要求处理器实现算法所描述的 store 语义。reservation 变量不仅会被其所属的处理器清空，也会被其他处理器清空。由于所有针对保留地址的写操作都会清空 reserved 旗标，所以“比较 - 比较并交换”操作可以据此来避免 ABA 问题，如算法 13.7 所示[⊖]。因此，LoadLinked/StoreConditionally 原语比 CompareAndSwap 更加强大，该原语允许开发者针对单个内存字实现任意类型的原子化“读 - 修改 - 写”操作。算法 13.8 展示了如何使用 LoadLinked/StoreConditionally 实现“比较并交换”、“比较并设置”原语。LoadLinked/StoreConditionally 原语还有另外一个特征需要注意：即使任何处理器都没有更新过保留地址的值，StoreConditionally 原语也有可能出现假性失败。多种低级硬件状况可能导致假性失败，其中值得注意的是中断的出现，包括缺页陷阱、溢出陷阱、时钟中断、I/O 中断等，这些中断均需要由内核进行处理。这种失败通常不会成为问题，但是如果 LoadLinked 和 StoreConditionally 之间的某段代码总是引发陷阱，则 StoreConditionally 操作可能始终都会失败。

238

算法 13.6 加载链接 / 条件存储语义

```
1 LoadLinked(address):
2     atomic
3     reservation ← address /* reservation 为每处理器变量 */
4     reserved ← true      /* reserved 为每处理器变量 */
5     return *address
6
7 StoreConditionally(address, value):
8     atomic
9     if reserved
10        store(address, value)
11        return true
12    return false
13
14 store(address, value):      /* 所有处理器均执行相同操作，无需同步 */
15     if address = reservation /* 判断粒度也可以是相同的高速缓存行或者其他 */
16        reserved ← false
17        *address ← value
```

算法 13.7 基于加载链接 / 条件存储实现原子性的状态迁移

```
1 observed ← LoadLinked(x)
2 计算 z 的新值
3 if not StoreConditionally(x, z)
4     返回并重新计算，或者解决冲突
```

算法 13.8 基于加载链接 / 条件存储实现比较并交换

```
1 compareAndSwapByLLSC(x, old, new):
2     previous ← LoadLinked(x)
3     if previous = old
```

⊖ 线程上下文切换也会导致 reservation 变量被清空。


```

4      StoreConditionally(x, new)
5      return previous
6
7  compareAndSetByLLSC(x, old, new):
8      previous  $\leftarrow$  LoadLinked(x)
9      if previous = old
10         return StoreConditionally(x, new)
11     return false

```

由于 LoadLinked/StoreConditionally 原语可以优雅地解决 ABA 问题，所以我们更加倾向于用其替代可能产生 ABA 问题的 CompareAndSwap 原语。当然，我们也可为 CompareAndSwap 原语关联一个计数器来解决 ABA 问题。

严格意义上讲，如果 StoreConditionally 原语所操作的并非之前保留的地址，则其最终结果可能是未定义的。但某些处理器在设计上便允许这种使用方式，这相当于提供了一种在某些场景下有用的、针对任意两个内存地址的原子操作。

239

13.3.3 原子算术原语

算法 13.9 定义了几种原子算术原语。我们也可使用 AtomicAdd 或 FetchAndAdd 原语来实现 AtomicIncrement 和 AtomicDecrement 操作，且其返回值既可以是原始值，也可以是新值。另外，处理器在执行这些原语时通常会设置条件码（condition code），其值可以用于反映目标地址的值是否为零（或者原始值为零），也可反映其他一些信息。在垃圾回收领域，FetchAndAdd 原语可以用于实现并发环境下的顺序分配（即阶跃指针分配），但更好的做法通常是每个线程建立本地分配缓冲区，如 7.7 节所述。FetchAndAdd 可以简单地用于从队列中添加或者移除元素的操作，但是对于环状缓冲区，还需要小心地处理回绕（wrap-around）问题（参见 13.8 节）。

算法 13.9 原子算术原语

```

1  AtomicIncrement(x):
2      atomic
3          *x  $\leftarrow$  *x + 1
4
5  AtomicDecrement(x):
6      atomic
7          *x  $\leftarrow$  *x - 1
8
9  AtomicAdd(x, v):
10     atomic
11         new  $\leftarrow$  *x + v
12         *x  $\leftarrow$  new
13         return new
14
15  FetchAndAdd(x, v):
16     atomic
17         old  $\leftarrow$  *x
18         *x  $\leftarrow$  old + v
19         return old

```

这些原子算术原语的能力严格弱于 CompareAndSwap，同时也弱于 LoadLinked/StoreConditionally（参见 Herlihy and Shavit [2008]）。每种原语都存在一个可以用一致数

(consensus number)[⊖]来描述的特征，如果某个原语的一致数为 k ，意味着它可以解决 k 个线程之间的一致问题，但无法解决多于 k 个线程之间的一致问题。所谓一致问题，是指多处理器算法是否能达到如下要求：①对于某个变量，每个线程均建议一个值；②所有线程针对某个值达成一致；③该变量的最终值为某个线程所建议的值；④所有线程均能够在有限步骤内完成操作，即算法必须满足无等待（wait-free）要求（参见 13.4 节）。对于所有的无条件设置原语（例如 AtomicExchange）或者对相同值产生相同运算结果的更新原语（如 AtomicIncrement 与 FetchAndAdd），其一致数均为 2。而 CompareAndSwap 和 LoadLinked/StoreConditionally 的一致数则为 ∞ ，即它们能够以无等待的方式解决任意多个线程之间的一致问题，正如算法 13.13 即将展示的。

无条件算术原语的一个潜在优势在于它们通常都会成功，而如果使用 CompareAndSwap 或者 LoadLinked/StoreConditionally 来模拟无条件算术原语，线程之间的竞争很可能导致“饥饿”现象的出现[⊖]。

13.3.4 检测 – 检测并设置

“检测 – 检测并交换”即为算法 13.3 中所介绍的 testAndTestAndSet。由于算法 13.3 会不断迭代，所以其正确性不存在问题。开发者应当避免将其实现为算法 13.10 所示的两种错误形式：testThenTestAndSetLock 不会进行迭代，如果 x 在 if 和 TestAndSet 语句之间被修改，TestAndSet 将执行失败，因而这一操作存在错误；testThenTestThenSetLock 的错误则更加明显，它不使用任何原子操作原语，因此在变量 x 的两次读取之间以及变量 x 的读写之间任何针对 x 的更新操作都有可能发生。需要注意的是，即使将 x 声明为 volatile 也无济于事。“比较 – 比较并交换”的实现也可能出现类似的错误。正确地构造出一个并发算法并非易事，这些错误便是开发者很容易遇到的陷阱。

算法 13.10 错误的“检测并设置”实现形式

```
1 testThenTestAndSetLock(x):                                /* 错误！ */
2     if *x == 0
3         TestAndSet(x)
4
5 testThenTestThenSetLock(x):                                /* 错误！ */
6     if *x == 0
7         其他工作
8         if *x == 0
9             *x ← 1
```

13.3.5 更加强大的原语

在我们描述过的硬件原语中，LoadLinked/StoreConditionally 的通用性最强，也是针对单字的最强的原子更新语义。除此之外，允许对多个独立字进行原子更新的硬件原语则更加强大。在单字原语之外，某些处理器还支持双字原语，例如双字比较并交换，我们称之为 CompareAndSwapWide/CompareAndSetWide（见算法 13.11）。如果仅从概念上来看，该原

⊖ 此处虽然也将 consensus 表述为“一致”，但其更倾向于“共识”。读者需要注意区分其与 coherence 和 consistency 的区别。——译者注

⊖ 如果竞争到达一定程度，甚至可能出现硬件层面的饥饿，即每个处理器都尝试以独占方式访问相关的高速缓存行。

语的强大之处没得到充分体现。但是,使用双字 CompareAndSwap 操作却可以轻松应对单字 CompareAndSwap 无法解决的 ABA 问题,此时我们只需要将第二个字用作记录第一个字被更新次数的计数器。对于 32 位的字,计数器的值最多可以达到 2^{32} ,因而基本上可以忽略计数器回绕可能带来的安全问题。支持对相邻两个 64 位的字进行原子更新的硬件原语则更加强大。因此,尽管 CompareAndSwapWide 在概念上与常规的 CompareAndSwap 并无较大差别,但其在使用上却更加方便、更加高效。

算法 13.11 CompareAndSwapWide

```

1 CompareAndSwapWide(x, old0, old1, new0, new1):
2     atomic
3         curr0, curr1  $\leftarrow$  x[0], x[1]
4         if curr0 = old0 && curr1 = old1
5             x[0], x[1]  $\leftarrow$  new0, new1
6         return curr0, curr1
7
8 CompareAndSetWide(x, old0, old1, new0, new1):
9     atomic
10        curr0, curr1  $\leftarrow$  x[0], x[1]
11        if curr0 = old0 && curr1 = old1
12            x[0], x[1]  $\leftarrow$  new0, new1
13        return true
14    return false

```

尽管更新相邻两个字的原子操作原语十分有用,但如果其能够原子化地更新内存中任意两个(不相邻)字则会显得更加强大。Motorola 880000 以及 Sun 的 Rock 处理器均提供了“双比较并交换”指令(compare-and-swap-two,也称 double-compare-and-swap),如算法 13.12 中的 CompareAndSwap2 原语所示。CompareAndSwap2 的硬件实现较为复杂,因而目前尚无商业级别的处理器支持这一原语。CompareAndSwap2 可以泛化为通用的 n 路比较并交换(compare-and-swap- n ,也称 n -way compare-and-swap),同理,也可对 LoadLinked/StoreConditionally 原语进行泛化,由此得到的结果便是事务内存(transactional memory),13.9 节将对其进行介绍。

算法 13.12 CompareAndSwap2

```

1 CompareAndSwap2(x0, x1, old0, old1, new0, new1):
2     atomic
3         curr0, curr1  $\leftarrow$  *x0, *x1
4         if curr0 = old0 && curr1 = old1
5             *x0, *x1  $\leftarrow$  new0, new1
6         return curr0, curr1
7
8 CompareAndSet2(x0, x1, old0, old1, new0, new1):
9     atomic
10        curr0, curr1  $\leftarrow$  *x0, *x1
11        if curr0 = old0 && curr1 = old1
12            *x0, *x1  $\leftarrow$  new0, new1
13        return true
14    return false

```

13.3.6 原子操作原语的开销

开发者经常会错误地使用原子操作原语,其原因之一是他们知道原子操作的开销较大,

因而会刻意避免使用原子操作，而另一种原因则是他们可能错误地使用了原子操作，例如算法 13.10 中两种错误的 `testAndTestAndSet` 实现。我们曾经提到，有两个原因导致原子操作的开销较大：一是原子化的“读-修改-写”原语必须以独占方式访问相关的高速缓存行；二是在指令结束之前，处理器必须完成数据读取、计算新值、写入新值这一系列操作。现代处理器可能会使用多指令重叠（`overlap`）技术，但是如果后续操作强依赖于原子操作的结果，则必然会减少流水线中的指令条数。由于原子操作需要确保一致性，因而其通常会涉及总线甚至内存的访问，这通常会花费较多的指令周期。

242

另一个导致原子操作执行速度较慢的原因是，它们要么天然包括内存屏障语义，要么要求开发者在其开始和结束位置手动添加额外的内存屏障。这潜在削减了指令重叠与流水线技术所带来的性能优势，从而导致处理器很难隐藏这些原语访问总线或者内存的开销。

13.4 前进保障

当多个线程之间竞争相同数据结构时（例如共享堆，或者回收器数据结构），确保整个系统能够正常往下执行尤为重要（特别是在实时环境中），我们将这一要求称为前进保障（`progress guarantee`）。了解不同硬件原语在前进保障方面的相对强度也十分必要，常见的前进保障级别从强到弱分别是：无等待、无障碍、无锁。对于并发算法而言，如果每个线程始终都可以向前执行（不论其他线程执行何种操作），则称其为无等待（`wait-free`）算法；如果在并发算法中，对于任意一个线程，只要其拥有足够长的独占式执行时间，便能够在有限步骤内完成操作，则称该算法为无障碍（`obstruction-free`）算法；如果算法永远可以保证某些线程能在有限步骤内完成操作，则称该算法为无锁（`lock-free`）算法。在真实系统中，前进保障通常是条件性的，例如，某一算法满足无等待要求的条件可能是存储空间尚未耗尽。Herlihy 和 Shavit [2008] 对这些概念及其实现进行了详尽的论述。

无等待算法通常会引入线程互助的概念，也就是说，如果线程 *t2* 即将执行的操作可能会打断线程 *t1* 正在执行的、在一定程度上可以确定超前于线程 *t2* 的操作，则线程 *t2* 将协助 *t1* 完成其工作，然后再开始执行自身工作。假设线程数量存在固定上界，且线程之间相互协助的工作单元或者对数据结构的操作也存在上界，则任何工作单元或者操作的完成步骤便都存在上界。但是，这一上界通常较大，且与较弱的前进保障相比，线程互助需要引入额外的数据结构与工作量，因而其操作时间通常相当长。对于较为简单的一致性场景，为其设计时间开销较小的无等待算法通常比较容易，如算法 13.13 所示。从中我们可以看出，该算法满足解决 *N* 个线程的一致问题的所有标准，但是其空间开销正比于 *N*。

算法 13.13 基于“比较并交换”原语来实现一致性（满足无等待要求）

```
1  shared proposals[N]                                /* 每个线程对应数组中的一个元素 */
2  shared winner ← -1                                  /* 优胜者线程编号 */
3  me ← myThreadId
4
5  decide(v):
6    proposals[me] ← v                                /* 0 ≤ 线程 id < N */
7    CompareAndSwap(&winner, -1, me)
8    return proposals[winner]
```

与无等待要求相比，无障碍更容易实现，但是其可能需要调度器的协助。如果线程发现当前存在竞争，则它可以使用随机递增的时间退让策略来确保其他线程优先完成工作。也就

243

是说，每当线程探测到竞争时，其首先会计算一个比上次退让时间更长的时间周期 T ，然后再从 $0 \sim T$ 之间选择某一随机值作为本次退让的时间。从概率上讲，对于较少出现竞争的场景，每个线程最终都会成功执行。

无锁要求的实现则更加简单，它只要求在任何情况下至少一个竞争者可以继续往下执行，即使其他线程可能会永久性地等待下去。

前进保障与并发回收

并行回收器 (parallel collector) 同时使用多个回收线程来处理回收工作，但其回收过程中仍会挂起所有赋值器线程；并发回收器 (concurrent collector) 会在赋值器线程执行的同时执行 (至少一部分) 回收工作，其通常也会使用多个回收线程。并行回收和并发回收算法的执行通常可以分为数个阶段，例如标记、扫描、复制、转发或清扫，并发回收器还可能会让赋值器在执行过程中承担一定的回收工作。多个回收线程之间必须进行协作，否则它们之间可能会相互干扰，或者干扰到赋值器的执行。在如此复杂的场景之下我们应当如何描述回收器的正确性？最基本的要求显然是回收器不能发生任何明显错误——至少要确保不会回收任何可达对象，并确保赋值器的正常工作。在此基础之上，回收器还应当确保回收工作终究能够结束，且其或多或少都应当回收一些不可达内存以便复用。对于不同的回收算法，其每次调用所能回收内存的期望值有所不同：保守式回收器 (只能依赖模糊根) 通常会高估堆中对象的可达性，进而导致某些不可达对象无法得到回收；类似地，对于分代回收器或者其他使用分区策略的回收器，其会故意避免回收堆中某些分区的不可达对象。完整的回收算法必须达到更高的要求：只要垃圾回收的调用次数足够多，任意垃圾对象最终都能得到回收。

并发回收器中还有其他一些问题需要关注。其中的一个问题是，对于在回收过程中 (由赋值器) 分配并 (在回收结束之前) 不可达的对象，或者在回收开始之前分配但在回收过程中不可达的对象，回收器是否应当将其回收。对于特定回收器而言，答案既可能是肯定的，也可能是否定的。

并发回收与并行回收过程中还存在更多微妙的问题与风险。顺序算法 (sequential algorithm) 的结束特征通常比较明显，例如在对可达对象图进行标记的过程中，堆中对象会被划分成 3 个集合：已标记且已扫描、已标记但未扫描、未标记，标记算法需要逐渐增大第一个集合，并最终使其囊括堆中所有可达对象。尽管算法在执行过程中有时难以达到严格的正确性要求，但判断算法何时执行结束却相对容易。在并发回收中，由于在回收过程中依然存在新对象的分配以及对象图的变更，所以算法的结束特征便不甚明显。如果赋值器的每一步操作都会产生更多的回收工作，那么回收器何时才能赶得上赋值器？为此，赋值器可能需要减缓执行甚至完全停止一段时间。即使可以确保回收器的处理速度永远高于赋值器，但仍存在一些其他的难点：除非回收算法使用无等待技术，否则回收器和赋值器之间的相互干扰可能会导致程序永远无法向前执行。例如，在无锁算法中，某一线程在尝试完成某个阶段的工作时可能会持续失败。同时，存在竞争关系的两个线程可能会永久性地导致对方无法向前执行，这一现象被称为活锁 (livelock)。

不同回收阶段的前进保障可能会有所不同——可能一个阶段为无锁级别，而另一个阶段为无等待级别。但在具体实现过程中，即使是在理论上完全无等待的算法也可能会引入一些 (期望停顿时间较短的) 万物静止式停顿。要达到最强的前进保障级别，不可避免地会增加代码复杂度并增加出错几率，因而在工程上花费大量精力以达到无等待要求的做法可能并不

值得。站在赋值器的角度来讲，只要回收器释放内存的速度足够快，能保证内存分配操作不会（由于等待回收器释放内存而）阻塞，就可以说回收算法达到无等待级别。另外，回收器还必须确保在回收周期结束之前堆空间不会耗尽。这些要求都远比在每个阶段中达到无等待要求要重要得多——它需要在堆大小、最大存活对象大小、分配率、回收率之间达到整体平衡。回收器的正常工作需要用足够的资源保证，包括内存与处理器时间。对于临界实时系统（critical real-time system）而言，这些因素都必须考虑，我们将在第 19 章提供更加详细的描述。后续章节中的大多数算法都仅提供较弱的前进保障，例如无锁保障，也可能只在特定的阶段才达到这一保障要求。较弱的前进保障不仅易于实现，而且在许多非严格场景也完全足够。

13.5 并发算法的符号记法

对于前面所讨论的几个问题，特别是在原子性、高速缓存一致性（cache coherence）、内存一致性（memory consistency）方面，开发者所编写的代码不一定会依照其表面上的顺序来执行——硬件与编译器可能会对某些操作进行重排序，甚至优化掉其中的某些操作。具体的执行结果在很大程度上取决于编程语言、编译器、运行时系统以及硬件。为屏蔽不同软硬件平台之间的差异，我们使用伪代码来描述算法。确保算法中某些操作的相对顺序对于算法的正确性十分重要，但并不是说所有操作都必须严格依照伪代码所示的顺序执行，且所有处理器都必须依照这一顺序感知到算法的状态变化。仅要求特定的代码序列必须有序执行，可以简化伪代码在具体环境中的实现。我们约定：

原子化的含义：位于 `atomic` 关键字范围之内的操作在发生的瞬间必须能够被所有处理器感知到——其执行过程可以看作是没有其他共享内存读写操作发生。当 `atomic` 操作发生冲突时（例如在某个处理器读取一个共享变量时，其他处理器同时读/写该变量），各处理器必须以相同的顺序感知到各操作的执行，且各线程在执行这些操作时也必须符合程序顺序。另外，`atomic` 代码段还必须能够充当其他共享内存访问操作的屏障，但由于并非所有硬件都会在原子操作原语中加入屏障语义，所以开发者可能需要手工添加内存屏障代码。此处的屏障代码可能会依照请求-释放（acquire-release）屏障语义进行工作，但我们在设计上假定其为完全内存屏障。

加载链接/条件存储的有序效应：加载链接和条件存储指令均必须能够充当共享内存访问的完全屏障。

变量标记：我们会对共享变量进行显式声明，其他变量均默认为线程私有变量。

数组：我们使用中括号来表示数组元素的数量，例如 `proposals[N]`。数组的声明会显式使用 `shared` 或者 `private` 关键字。数组可以使用元组进行初始化，例如 `shared pair[2] ← [0, 1]`，元组也可使用相同的元素扩展到特定长度，例如 `shared level[N] ← [-1, ...]`。

指向共享变量的引用：我们假定对共享变量的每次引用都会引发一次真正的内存读写操作，但其发生顺序并不一定要与代码顺序保持一致。

因果性：如果操作 x 和操作 y 之间存在因果联系，则在真实系统中 x 应当先于 y 执行，即执行结果应当满足伪代码所展示的顺序语义。此处的一个例子是依赖性的内存引用：如果代码需要访问由共享指针变量 p 所引用对象中的某个域 f ，即 $(*p).f$ ，则读取 p 的操作和读取 f 的操作之间存在因果联系。类似的案例还包括通过共享索引变量 i 访问共享数组中的元素，即 $a[i]$ 。

条件控制语句同样也隐含着因果性要求：**if**、**while** 或者其他条件控制表达式的运算结果会因果性地决定后续代码的执行路径，因此开发者必须小心地禁止条件代码的臆测求值 (speculative evaluation)，进而避免处理器对共享变量的访问进行重排序。但是，**if** 语句之后的非条件性代码与 **if** 表达式之间并不存在因果联系，类似的情况还包括将条件代码移出循环。

显式屏障点：即使算法的具体实现完全遵守上述各项约定，编译器或者硬件仍可能对许多操作进行任意排序，但在某些情况下，算法仍需要按特定的顺序执行以确保结果的正确性。因此我们在约定中补充一条：所有结尾带 “\$” 符号的代码都必须依照算法所示的顺序执行。另外，带 “\$” 符号的行同时也可以视作共享内存访问的完全屏障。本章目前为止所出现的代码都不需要引入 “\$” 标记。需要注意的是，对于带 “\$” 符号的行的具体实现，某些处理器架构可能需要在该行之前添加某种屏障，也可能是在其后添加，也可能是前后都需要添加。这些行通常都是比较重要的、不允许重排序的特殊读 / 写操作。尽管 “\$” 标记并不能对伪代码在具体平台上的实现给予完整指引，但它却指明了哪些地方需要特别注意。

13.6 互斥

互斥是并发计算中最基本的问题之一，开发者可以借助互斥机制来确保某一代码段在任意时刻只会被一个线程执行，此处的代码段被称为临界区 (critical section)。尽管在某些情况下我们可以借助一条原子操作原语来实现必要的状态转换，也可能使用具有较强前进保障级别的技术，但这可能会引入很大的开销，并且几乎必然会增大代码复杂度。因此在许多场景下，互斥技术依然十分有用。借助于原子化的 “读 - 修改 - 写” 原语很容易实现加锁 / 解锁函数，如算法 13.1 ~ 13.3 所示。但即使不借助于原子操作原语，我们仍可以借助于 (以适当方式排序的) 共享内存读写操作实现互斥。算法 13.14 展示了一种经典的互斥技术，即 Peterson 算法，该算法可以实现两个线程之间的互斥，其不仅能够正确实现互斥，而且满足前进保障要求，即如果两个线程同时请求进入临界区，必然有一个线程可以成功。该算法中，线程的等待存在上界，即线程在成功进入临界区之前所需等待的轮数存在上界^①。对于算法 13.14 而言，需要等待的轮数上界为 1，即等待当前占据临界区的线程离开临界区。

算法 13.14 基于 Peterson 算法实现互斥

```

1  shared interested[2] ← [false, false]
2  me ← myThreadId
3
4  petersonLock():
5      other ← 1 - me                                /* 线程 id 必须为 0 或 1 */
6      interested[me] ← true
7      victim ← me                                     $
8      while victim = me && interested[other]         $
9          /* 空等待 */
10
11 petersonUnlock():
12     interested[me] ← false

```

Peterson 算法很容易改造成适用于 N 线程的互斥算法，如算法 13.15 所示，我们可以从中看出其与双线程实现版本的相似之处。**while** 循环的设计十分巧妙。该算法将竞争临界区

^① 等待时间存在上界的前提是：进入临界区的线程能够在有限时间内离开临界区。

的线程分为 N 个等级，其基本思想在于，如果某一线程发现不存在同级别或者更高级别的线程，则其可以尝试向前晋升一个级别。但是，对于处于某一级别的线程，如果另一个线程也到达同一级别，则后者会更改 `victim` 变量并更早得到晋升。也就是说，只有最晚到达特定级别的线程才有资格等待更高级别的线程得到晋升（或进入临界区），一旦该线程得到晋升，则同级别或更低级别的其他线程将会立刻感知到。该算法中，即使 `while` 循环条件变量使用非原子化的方式计算也能保证算法的正确性。Peterson 算法展示了基于非原子操作来实现互斥的可行性，但原子操作原语却更加方便、更加实用。

246

算法 13.15 适用于 N 线程互斥的 Peterson 算法

```
1 shared level[N] ← [-1,...]
2 shared victim[N]
3 me ← myThreadId
4
5 petersonLockN():
6   for lev ← 0 to N-1
7     level[me] ← lev                      /* 0 ≤ 线程 id < N */
8     victim[lev] ← me                      $
9     while victim[lev] = me && (∃i ≠ me)(level[i] ≥ lev)  $
10      /* 空等待 */
11
12 petersonUnlockN():
13   level[me] ← -1
```

在 13.3 节对一致问题（consensus problem）的描述中，我们介绍了一致问题的无等待解决方案（即算法 13.13）。如果没有较强的前进保障要求，则通过互斥机制也可简单地解决一致性问题，如算法 13.16 所示。Peterson 算法可以实现互斥，因而它也可以用于解决此类一致性问题。但是，如果可以使用 `CompareAndSwap` 原语，则其通常是更加合适的解决方案（如算法 13.13 所示）。

算法 13.16 基于互斥来实现一致性

```
1 shared winner ← -1
2 shared value
3 me ← myThreadId
4
5 decideWithLock(v):
6   lock()                                /* 简单但无法提供较强的前进保障 */
7   if winner = -1
8     winner ← me
9     value ← v
10  unlock()
11  return value
```

13.7 工作共享与结束检测

并行回收与并发回收算法通常都需要通过某种方式来检测并行算法的结束（termination）。需要注意的是，这一概念与论证并行算法最终是否可以结束有着本质的区别，此处我们所关注的是如何检测并行算法的一个特定实例是否真正执行完毕。例如对于各回收线程“获取工作、处理工作单元、产生更多工作”这种普遍的并行回收模式，如果每个线程只关注自身的工作，则结束检测便十分简单——线程在工作结束之后设置本地 `done` 旗标，如果所有线程的

旗标都已设置,则算法结束。但是,并行算法通常都会以某种方式共享工作单元,其目的是对线程之间的工作负载进行平衡,从而最大化地利用多处理器、提升处理速度。这种平衡可以通过两种方式实现:一是工作负载相对较重的线程可以将部分工作推送(push)给负载较轻的线程,二是工作负载较轻的线程从负载较重的线程中拉取(pull)部分工作。工作拉取策略也称为工作窃取(work stealing)。

线程之间的工作迁移必须原子化地进行,或者至少应当确保任何工作单元都不会丢失[⊖],但此处我们主要关注的是如何实现工作共享算法的结束检测。我们可以使用一个共享的计数器来表示所有剩余工作单元的数量,同时每个线程原子化地对该计数器进行更新,进而实现一种简单的结束检测机制。但如果多个线程对共享计数器的更新过于频繁,则计数器本身很可能成为性能瓶颈[⊖]。因此许多结束检测算法都会避免使用原子更新原语,同时避免将读写操作集中在单个变量上。最容易想到的一种解决方案是使用独立的线程来检测其他线程的工作是否完成。

算法 13.17 展示了 Leung 和 Ting[1997] 所提出的共享内存工作共享结束算法的简化版本[⊖],该算法针对工作推送模式而设计。该算法的基本思想是,每个工作线程使用一个 busy 旗标来表示其是否在工作中,检测线程需要对这些旗标进行扫描。需要注意的是,如果空闲工作线程收到了来自其他线程的工作推送,则其将重新变为工作状态。但在进行结束检测时,发起工作推送的线程可能已经完成了自身的工作并转入空闲状态。由于检测线程无法原子性地完成所有 busy 旗标的扫描,所以其可能会先发现工作接收线程处于空闲状态(由于工作推送尚未开始),然后又发现工作推送线程也处于空闲状态(由于工作推送已经完成),此时检测线程可能产生错误的结束判定。为解决这一问题,算法使用 jobsMoved 变量来表示当前是否有工作迁移发生,当该变量被设置时检测线程会重新进行检测。需要特别注意的是,sendJobs 方法会等待 busy[j] 被设置为真时才进行下一步操作(第 24 行),这同样是为了避免检测线程产生误判。只有当所有工作都已经处理完毕,所有线程的 busy 旗标才可能全部都为 false。

算法 13.17 $\alpha\beta\gamma$ 共享内存终结算法的简化版本 [Leung and Ting, 1997]

```

1  shared jobs[N] ← 各线程的初始工作
2  shared busy[N] ← [true,...]
3  shared jobsMoved ← false
4  shared allDone ← false
5  me ← myThreadId
6
7  worker():
8      loop
9          while not isEmpty(jobs[me])
10             if the job set of some thread j appears relatively smaller than mine

```

⊖ 某些情况下工作单元是幂等(idempotent)的,因而即使多次处理该工作单元,算法的正确性依然可以得到保证(尽管这可能浪费部分计算资源)。

⊖ Flood 等[2001]在其并行回收器中使用的方案是:利用一个字中的每一位来表示每个线程的工作是否完成,当该字为零时意味着算法结束。该方案与此处所描述的方案在本质上并无差别。

⊖ 为简化表述,我们省略了 Leung 和 Ting[1997]中的 β 旗标,该旗标用于表示线程是处于休眠状态还是唤醒状态。我们同时为 α 和 γ 旗标赋予了更容易记忆的名称 busy 和 jobsMoved。Leung 和 Ting 同时还展示了该算法的一个变种,改进后的算法在每经历 \sqrt{N} 次迭代之后才需要检测 jobsMoved 旗标。与回收算法中执行回收任务所需的时间相比,这一改进的实际意义值得怀疑。

```

11         some ← chooseAndDequeueJobs()
12         sendJobs(some, j)                                $
13     else
14         job ← dequeue(jobs[me])
15         perform job
16         busy[me] ← false                                  $
17         while isEmpty(jobs[me]) && not allDone            $
18             /* 等待算法结束或者其他线程推送工作到本线程 */
19         if allDone return                                  $
20         busy[me] ← true                                    $
21
22 sendJobs(some, j):                                       /* 将工作推送到负载更轻的线程 */
23     enqueue(jobs[j], some)                                $
24     while (not busy[j]) && (not isEmpty(jobs[j]))          $
25         /* 等待线程 j 被唤醒 */
26         /* 部分工作发生转移 */
27         jobsMoved ← true                                  $
28
29 detect():
30     anyActive ← true
31     while anyActive
32         anyActive ← (∃i)(busy[i])
33         anyActive ← anyActive || jobsMoved                $
34         jobsMoved ← false                                  $
35     allDone ← true                                        $

```

算法 13.18 展示了另一种工作共享策略——工作窃取（拉取）模式下的相似算法，Endo 等 [1997] 在其并行回收器中使用的结束检测算法本质上即为该算法。尽管 Herlihy 和 Moss [1992] 的无锁回收器并未使用工作共享技术，但其结束算法的核心与算法 13.18 中 `busy` 和 `jobsMoved` 旗标的相关逻辑是一致的。

算法 13.18 工作窃取模式下的 $\alpha\beta\gamma$ 式结束检测算法

```

1 me ← myThreadId
2
3 worker():
4     loop
5         while not isEmpty(jobs[me])
6             job ← dequeue(jobs[me])
7             perform job                                $
8         if 某一线程 j 的工作负载相对较重
9             some ← stealJobs(j)                        $
10            enqueue(jobs[me], some)
11            continue
12            busy[me] ← false                              $
13            while no thread has jobs to steal && not allDone    $
14                /* 等待算法结束 */
15            if allDone return                                  $
16            busy[me] ← true                                    $
17
18 stealJobs(j):
19     some ← atomicallyRemoveSomeJobs(jobs[j])
20     if not isEmpty(some)
21         jobsMoved ← true                                  /* 部分工作发生转移 */
22     return some

```

我们还可以对检测算法进行改进，即只有当某一线程处于空闲状态时才进行结束检测，

在此之前检测线程仅需要关注旗标 `anyIdle`，如算法 13.19 所示。类似地，在工作窃取模式下，我们也可使用相同的策略来改进算法，即检测线程（在检测 `allDone` 之前）先专注于旗标 `anyLarge` 的检测，如算法 13.20 所示。

算法 13.19 直到有线程处于空闲状态时才进行结束检测

```

1  shared anyIdle ← false
2  me ← myThreadId
3
4  worker():
5      ...
6      busy[me] ← false                                $
7      anyIdle ← true                                  $
8      ...
9
10 detect():
11     anyActive ← true
12     while anyActive
13         anyActive ← false
14         while not anyIdle                                $
15             /* 等待有线程结束工作 */
16             anyIdle ← false                                $
17             anyActive ← (∃i)(busy[i])                    $
18             anyActive ← anyActive || jobsMoved            $
19             jobsMoved ← false                            $
20     allDone ← true                                        $

```

算法 13.20 延迟空闲工作线程的结束检测

```

1  shared anyLarge ← false
2  me ← myThreadId
3
4  worker():
5      loop
6          while not isEmpty(jobs[me])
7              job ← dequeue(jobs[me])
8              perform(job)                                $
9              if 本线程的工作负载较重
10                 anyLarge ← true                            $
11          if anyLarge
12              anyLarge ← false /* 在查看其他线程的负载之前先将 anyLarge
13                                     设置为 false */
14              if another thread j has a relatively large jobs set
15                 anyLarge ← true /* 可能存在更多可窃取工作 */
16                 some ← stealJobs(j)
17                 enqueue(jobs[me], some)
18                 continue
19          busy[me] ← false                                $
20          while (not anyLarge) && (not allDone)
21              /* 等待算法结束 */
22              if allDone return                                $
23          busy[me] ← true                                    $

```

目前为止我们所介绍的结束检测算法都需要一个额外的结束检测线程，我们也可使用空闲线程来进行结束检测，如算法 13.21 所示。但不幸的是，该算法并不能正常工作：假设线程 A 在完成自身工作之后发现没有工作可以窃取到，其将进行结束检测。在其检测扫描过程中，它可能会发现线程 B 出现了额外的工作，因而其将中止结束检测过程并准备设置自

身的 busy 旗标。但在此时，线程 B 却完成了自身的所有工作并转入结束检测状态，它将发现所有线程都已完成工作，进而判定算法结束。这一问题最简单的解决方案是在进行结束检测时使用互斥机制，如算法 13.22 所示。

算法 13.21 对称结束检测

```
1  work():
2      ...
3      while 本线程工作结束 && not allDone
4          /* 此算法存在问题！ */
5          detectSymmetric()
6      ...
7
8  detectSymmetric():
9      while not allDone
10         while (not anyIdle) && (not anyLarge)
11             /* 一直等待到结束检测可能成功为止 */
12             if anyLarge return
13             anyIdle ← false
14             anyActive ← (∃i)(busy[i])
15             anyActive ← anyActive || jobsMoved
16             jobsMoved ← false
17             allDone ← not anyActive
```

算法 13.22 修正后的对称结束检测

```
1  shared detector ← -1
2  me ← myThreadId
3
4  work():
5      ...
6      while I have no work && not allDone
7          if detector ≥ 0
8              continue /* 避免多个线程同时进行结束检测 */
9          if CompareAndSet(&detector, -1, me)
10             detectSymmetric()
11             detector ← -1
12      ...
```

为了更加全面地介绍各种结束检测策略，算法 13.23 展示了使用原子更新计数器来实现结束检测的方法。我们将在 13.8 节介绍一种能够支持工作共享的无锁数据结构——并发双端队列（double-ended queue）。

算法 13.23 基于计数器的结束检测

```
1  shared numBusy ← N
2  worker():
3      loop
4          while work remaining
5              perform(work)
6          if AtomicAdd(&numBusy, -1) = 0
7              return
8          while 无工作可以窃取 && (numBusy > 0)
9              /* 等待新工作的到来，或者算法结束 */
10             if numBusy = 0
11                 return
12             AtomicAdd(&numBusy, 1)
```

汇聚屏障

并发回收或者并行回收中另一种常见的同步机制是要求所有回收线程到达算法的某一点（基本上就是某一回收阶段的结束点），然后再继续往下执行。一般情况下，上述任何一种结束检测算法都可以胜任这一场景的要求。另一种常见的场景是：算法在某一阶段的工作并不存在任何形式的工作共享或者负载均衡，但其仍要求所有线程都到达指定点，即汇聚屏障（rendezvous barrier）。此时便可使用计数器结束检测算法（即算法 13.23）的简化版本，如算法 13.24 所示。在程序执行过程中，回收器通常会被调用多次，因而要么算法需要在开始时重置计数器的值，要么就要求在汇聚计数器被重置时，任何线程都不会依赖该值。算法 13.25 使用了后一种技术来重置计数器。

算法 13.24 使用计数器实现线程汇聚

```

1  shared numBusy ← N
2
3  barrier():
4      AtomicAdd(&numBusy, -1)
5      while numBusy > 0
6          /* 等待其他线程到达汇聚点 */

```

算法 13.25 具有计数重置能力的汇聚屏障

```

1  shared numBusy ← N
2  shared numPast ← 0
3
4  barrier():
5      AtomicAdd(&numBusy, -1)
6      while numBusy > 0
7          /* 等待其他线程到达汇聚点 */
8          if AtomicAdd(&numPast, 1) = N /* 只有一个优胜线程可以重置计数器 */
9              numPast ← 0                $
10             numBusy ← N                $
11         else
12             while numBusy = 0 /* 其他线程等待计数器得到重置（但不会等待很久）*/
13                 /* 等待计数器重置完成 */

```

13.8 并发数据结构

我们有必要对并行回收器和并发回收器中经常用到的一些数据结构进行介绍，并剖析对应的具体实现技术。针对顺序程序设计的数据结构通常无法胜任并行系统或并发系统的要求，它们很容易遭到破坏。如果某一数据结构很少得到访问，则可以简单地使用互斥机制，即为该数据结构的每个实例创建一个锁，任何线程在对其进行操作之前都必须先获取锁，并在操作完成后释放锁。如果存在内嵌操作或者递归操作，则可以使用计数锁（counting lock），如算法 13.26 所示。

算法 13.26 计数锁

```

1  /* 锁本身占据一个字的空间，其内部包含两个信息：线程 ID 以及计数器 */
2  shared lock ← {thread: -1, count: 0}int
3  me ← myThreadId
4

```

```

5  countingLock():
6      old ← lock
7      if old.thread = me && old.count > 0
8          /* 仅增加计数 (假设不会出现溢出) */
9          lock ← {old.thread, old.count + 1}
10         return
11     loop
12         if old.count = 0
13             if CompareAndSet(&lock, old, {thread: me, count: 1})
14                 return
15         old ← lock
16
17 countingUnlock():
18     /* 线程释放锁, 即使计数变为零, 算法依然正确 */
19     old ← lock
20     lock ← {old.thread, old.count - 1}

```

对于某些需要频繁访问的数据结构, 简单的互斥机制可能会成为瓶颈, 因此研究者们提出了多种并发数据结构, 这些数据结构允许并发操作之间存在更大的重叠 (overlap)。即使并发操作发生重叠, 仍可以保证操作的安全性与结果的正确性。对于某一数据结构而言, 如果任意两个发生重叠的操作对数据结构造成的状态变化与响应结果与它们不发生重叠时的执行结果相同, 则称该数据结构是线性化 (linearisable) 的 [Herlihy and Wing, 1990]。另外, 如果两个操作在时间上不重叠, 则它们的执行顺序看起来必须与其调用顺序保持一致。对于每一个操作, 我们都可以认为其在某一时刻发生, 并称该时刻为线性化点 (linearisation point)。一个操作在时间上可能存在多个线性化点, 但是对于多个相互影响的操作, 其线性化点之间的相对顺序应当与各操作的逻辑顺序保持一致。如果各操作之间不会相互影响, 则它们必然能够满足线性化的要求。许多内存管理操作 (例如内存分配与工作列表变更) 必须遵从线性顺序。

[253]

有多种通用实现策略可供开发者在构建并发数据结构时选择, 这些策略在并发程度方面从最低到最高 (通常也是从最简单到最复杂) 分别如下^①:

粗粒度锁 (coarse-grained locking): 使用一个“大”锁来控制整个数据结构的访问 (如上面提到的互斥方案)。

细粒度锁 (fine-grained locking): 该方案为较大数据结构中的每个元素维护独立的锁, 例如为链表或树中的每个节点维护一个锁。如果各线程对数据结构的访问与更新足够分散, 则该策略可以显著提升并发程度。该策略通常需要考虑一个问题, 即如果某一操作会对多个元素加锁, 则其必须保证没有其他相同操作 (或者任何其他操作) 会以相反的顺序对相同元素进行加锁, 否则将导致死锁的产生。一种通用的解决方案是要求所有操作在访问 (单链表或者树中的) 元素时遵从相同的方向, 即锁联结 (lock coupling): 某一操作先对节点 A 加锁, 然后再对 A 所指向的节点 B 加锁, 接着再释放节点 A 的锁并对节点 B 所指向的节点 C 加锁, 以此类推。使用这一逐步推进策略来遍历数据结构可以确保后发线程无法超越先发的线程, 同时也可以确保在链表 / 树中插入 / 删除元素操作的安全性。细粒度锁的一个潜在缺陷是, 对不同元素多次加解锁可能会给共享总线或者存储带来一定开销, 进而掩盖了相对于粗粒度锁的优势。

[254]

乐观锁 (optimistic locking): 该方案对细粒度锁进行了改进, 即线程在对数据结构进行

① Herlihy 和 Shavit [2008] 文献第 9 章详细介绍了在每种并发级别下如何以链表形式实现集合。

遍历时先不加锁，直到其找到合适元素时才尝试对其进行加锁。但在从发现合适元素到加锁成功的过程中，该元素可能已被其他并发线程修改，因而线程在加锁完成后需要再次对该元素进行校验。如果校验失败，则其需要释放锁并继续查找。这种尽量将加锁操作延迟、直到迫不得已时才加锁的策略可以减少开销并提升并发能力。乐观锁在大多数场景下都具有较高的性能，但如果频繁发生更新冲突，则会导致性能下降。

懒惰更新 (lazy update)：即使采用乐观锁，只读操作仍需要对其所读取的元素加锁，这不仅会成为提升并发程度的瓶颈，还可能在只读操作中引入写操作（即加解锁）。为此我们可以设计出一种数据结构，在该结构中只读操作无需加锁，代价是更新操作的复杂度稍有提高。一般来讲，懒惰更新策略中的写操作需要先达到其在“逻辑上”的目标（前提是不影响其他线程的并发访问），然后再进一步执行真正的更新操作，并确保数据结构回归正常状态。我们可以通过一个例子来理解懒惰更新的含义：对于以链表方式实现的集合，`remove` 操作首先需要（在逻辑上）将待移除元素打上 `deleted` 标记，然后再将其前一个节点的指针重定向，从而真正实现节点的移除。为避免并发更新可能带来的问题，所有操作都需要在持有相关元素锁的前提下执行。`remove` 操作必须依照先标记再移除的顺序执行，这样才能确保其他线程能够在不加锁的情况下正确进行读操作。向链表中插入元素的操作只需要更新数据结构中的 `next` 指针，因而只需要一次更新操作（即无需事先设置标记），当然，这一操作同样必须在持有相关元素锁的前提下执行。

非阻塞 (non-blocking)：在非阻塞策略中，对数据结构的所有操作均无需使用锁，仅依赖原子更新原语便可完成数据结构的状态变更。一般来说，状态变更操作通常都会存在某些特殊的原子更新事件，这些事件的发生点即为该操作的线性化点。与此相比，基于锁的策略则需要引入临界区来标识线性化“点”^①。非阻塞策略的并发能力可以通过前进保障来衡量，如 13.4 节所述：无锁实现可能允许部分线程出现饥饿；无障碍实现中的每个线程可能需要足够长的时间进行独占式操作，才能确保算法整体往下执行；无等待实现可以确保所有线程都能正确往下执行。三种前进保障在实现上的难度依次增大。本章的后续部分简要介绍了部分无锁算法实现，其无等待实现版本可参见 Herlihy 和 Shavit [2008]。

本节后续部分将简单介绍与并行回收和并发回收相关性较高的算法，算法的具体实现基本都遵循 Herlihy 和 Shavit 的建议。

255

13.8.1 并发栈

我们首先介绍如何基于单链表来实现并发栈。由于栈的可操作位置只有一处，所以各种基于锁的策略在性能上差异不大，其实现代码也比较简单，我们不在此逐一列出。算法 13.27 展示了一种栈的无锁实现方式，其 `push` 操作的无锁实现相对容易，但 `pop` 操作则稍微复杂一些。`popABA` 是 `pop` 操作的一种简单实现方案，它基于 `CompareAndSet` 原语，但其无法解决 ABA 问题，即：在当前线程执行 `pop` 操作时，如果其他线程将 `currTop` 所引用的节点弹出，然后该节点在某一时刻又被重新压入栈中，且其 `next` 指针与 `currTop.next` 不同，则当前线程可能会误认为栈尚未发生变化。算法 13.27 同时还展示了基于 `LoadLinked/StoreConditionally` 或者 `CompareAndSetWide` 原语的 `pop` 实现方案，它们都可以解决 ABA 问题。

① 由于临界区需要依赖互斥机制，所以只有当其他并发操作也尝试进入临界区时，临界区才能称为线性化点。懒惰更新策略通常存在单个线性化点。

算法 13.27 基于单链表的无锁栈实现

```

1  shared topCnt[2] ← [null, any value]
2  shared topAddr ← &topCnt[0]                                /* 栈顶 */
3  shared cntAddr ← &topCnt[1]                                /* 仅供 popCount 使用的变更计数 */
4
5  push(val):
6      node ← new Node(value: val, next: null)
7      loop
8          currTop ← *topAddr
9          node.next ← currTop
10         if CompareAndSet(topAddr, currTop, node)
11             return
12
13  popABA():
14      loop
15          currTop ← *topAddr
16          if currTop = null
17              return null
18          /* 如果 currTop 所指向的节点得到复用, 则可能产生 ABA 问题 */
19          next ← currTop.next
20          if CompareAndSet(topAddr, currTop, next)
21              return currTop.value
22
23  pop():
24      loop
25          currTop ← LoadLinked(topAddr)
26          if currTop = null
27              return null
28          next ← currTop.next
29          if StoreConditionally(topAddr, next)
30              return currTop.value
31
32  popCount():
33      loop
34          currTop ← *topAddr
35          if currTop = null
36              return null
37          currCnt ← *cntAddr
38          nextTop ← currTop.next
39          if CompareAndSetWide(&topCnt, currTop, currCnt,
40                               nextTop, currCnt+1)
41              return currTop.value

```

对于基于数组的并发栈，最好的实现方案是使用锁。并发栈通常都会存在性能瓶颈，这不仅是由于各处理器必须保证高速缓存和内存的一致性，而且因为所有操作都必须串行化。针对这一问题存在多种解决方案。Blelloch 和 Cheng [1999] 提出了一种无锁解决方案，该方案要求所有线程的并发操作要么必须全部是压入操作，要么必须全部是弹出操作，从而可以通过 FetchAndAdd 原语而非锁来控制栈顶指针，我们将在第 14 章详细介绍这一算法。Herlihy 和 Shavit 的文献的第 11 章介绍了一种并发无锁栈实现方案，在该策略中，线程在竞争情况比较严重时会尝试在额外的缓冲区中完成操作，即如果弹出操作发现待压入操作，或者压入操作发现待弹出操作，则压入操作将立刻满足弹出操作：两个操作相互抵消。这一操作发生的瞬间（必然是压入操作在前、弹出操作在后）满足线性化要求，同时该操作不会受到主体栈中任何操作的影响。

13.8.2 基于单链表的并发队列

并发队列存在两个可操作位置，元素从队首出队，从队尾入队，因而其比并发栈具有更高的并发化价值。我们可以为队列设置一个“虚拟”节点，该节点位于下一个即将出队的元素之前。head 指针指向虚拟节点，tail 指针指向最后入队的元素。如果队列为空，则 tail 也指向虚拟节点。

算法 13.28 展示了基于细粒度锁的并发队列实现，每个可操作位置对应一个锁。需要注意的是，remove 操作将 head 指针重定向到虚拟节点的下一个节点，因而当首次执行 remove 成功后，原始的虚拟节点将被释放，而包含刚刚出队的值的节点则将成为新的队首虚拟节点。这一版本的 Queue 无法限制队列的最大长度。算法 13.29 所展示的 BoundedQueue 也使用类似机制实现，但它能够限制队列的最大长度。为避免将队列长度变更操作集中在变量 size 域而引发竞争，该算法分别维护了整个队列入队操作和出队操作的次数。即使这两个值出现回绕也不会出现问题，因而只要确保用于存储这两个值的域能够表示从 0 到 MAX 之间的 MAX + 1 个整数即可。另外，如果这两个计数变量位于同一个高速缓存行中，则该算法的性能将不会优于仅使用单个 size 域的策略。

算法 13.28 基于单链表的细粒度锁并发队列

```

1  shared head ← new Node(value: dontCare, next: null)
2  shared tail ← head
3  shared addLock ← UNLOCKED
4  shared removeLock ← UNLOCKED
5
6  add(val):
7      node ← new Node(value: val, next: null)
8      lock(&addLock)
9      tail.next ← node
10     tail ← node
11     unlock(&addLock)
12
13  remove():
14      lock(&removeLock)
15      node ← head.next
16      if node = null
17          unlock(&removeLock)
18          return EMPTY          /* 队列为空 */
19      val ← node.value
20      head ← node
21      unlock(&removeLock)
22      return val

```

算法 13.29 基于单链表的细粒度锁有界并发队列

```

1  shared head ← new Node(value: dontCare, next: null)
2  shared tail ← head
3  shared addLock ← UNLOCKED
4  shared removeLock ← UNLOCKED
5  shared numAdded ← 0
6  shared numRemoved ← 0
7
8  add(val):
9      node ← new Node(value: val, next: null)
10     lock(&addLock)

```

```

11  if numAdded - numRemoved = MAX
12      unlock(&addLock)
13      return false                /* 队列已满 */
14  tail.next ← node
15  tail ← node
16  numAdded ← numAdded + 1        /* 即使发生整数回绕，算法依然正确 */
17  unlock(&addLock)
18  return true                    /* 入队成功 */
19
20  remove():
21      lock(&removeLock)
22      node ← head.next
23      if numAdded - numRemoved = 0
24          unlock(&removeLock)
25          return EMPTY            /* 队列为空 */
26      val ← node.value
27      head ← node
28      numRemoved ← numRemoved + 1 /* 即使发生整数回绕，算法依然正确 */
29      unlock(&removeLock)
30      return val

```

上述并发队列还具有一个重要的特征：如果只有一个线程可以执行入队（或者出队）操作，则入队（或者出队）操作便无需加锁。特别地，如果能够执行入队和出队操作的线程各仅有一个，则整个算法可以完全不需要锁。垃圾回收算法中常见的一种场景是存在多个入队线程和一个出队线程，尽管其仍需要在队尾加锁，但是与在队列两端都需要加锁的一般算法相比，该算法显然更加优秀。

对于基于单链表的并发队列，其他基于锁的策略（例如乐观锁或者懒惰更新）在本质上并不能将队列的并发能力提升到比细粒度锁更高的程度。

算法 13.30 展示了基于单链表并发队列的无锁实现方案。该方案的巧妙之处在于其入队操作需要两步完成：首先需要将当前的队尾节点更新到新节点，然后再将 `tail` 指针更新到新节点。无锁算法必须假定其他正在执行入队（或者出队）操作的线程能够感知到入队的中间状态。该算法解决这一问题的方案是：任何线程在发现 `tail` 指针出现“不同步”时将其修正，因此任何线程均无需等待其他线程完成这一操作，这便是无等待算法中常用的互助（help）策略。但是，该算法本身却达不到无等待的前进保障级别。

算法 13.30 基于单链表并发队列的无锁实现

```

1  shared head ← new Node(value: dontCare, next: null)
2  shared tail ← head
3
4  add(val):
5      node ← new Node(value: val, next: null)
6      loop
7          currTail ← LoadLinked(&tail)
8          currNext ← currTail.next
9          if currNext ≠ null
10             /* tail 指针出现不同步，尝试帮助其修复 */
11             StoreConditionally(&tail, currNext)
12             continue                /* 在尝试修复之后重新尝试入队 */
13         if CompareAndSet(&currTail.next, null, node)
14             /* 入队，并尝试更新 tail 指针 */
15             StoreConditionally(&tail, node)
16             /* 即使失败也可以接受，因为其他线程已经（或者将会）把 tail 更新到同步状态 */

```

```

17         return
18
19     remove():
20         loop
21             currHead ← LoadLinked(&head)
22             next ← currHead.next
23             if next = null
24                 if StoreConditionally(&head, currHead)
25                     /* head 未发生变化, 因而队列必然为空 */
26                     return EMPTY /* 队列为空 */
27                     continue /* head 可能发生变化, 重试 */
28
29             currTail ← tail
30             if currHead = currTail
31                 /* 第 24 行判断队列非空, 而此处又判断队列为空, 发生不同步, 帮助修复 */
32                 currTail ← LoadLinked(&tail)
33                 next ← currTail.next
34                 if next ≠ null
35                     StoreConditionally(&tail, next)
36                 continue
37
38             /* 队列非空, 且状态同步, 尝试移除首节点 */
39             val ← next.value
40             if StoreConditionally(&head, next)
41                 return val
42             /* 失败则重试 */

```

13.8.3 基于数组的并发队列

与基于链表的队列相比, 基于数组的队列不仅具有更高的存储密度, 而且不必动态分配节点。有界队列可以使用环状缓冲区的方式实现, 算法 13.31 即展示了使用细粒度锁的环状缓冲区。我们也可对其进行改进: 省略 numRemoved 和 numAdded 变量, 并通过 tail 和 head 之间的求差取模运算来获取队列长度, 如算法 13.32 所示。当 MAX 为 2 的整数次幂时算法性能最佳, 因为此时取模操作可以通过位掩码操作完成。引入 MODULUS 变量的原因在于, tail 和 head 之间的距离 (即缓冲区中元素的数目) 存在 MAX + 1 种可能, 因此取模操作的模数必须大于 MAX。与此同时, 为确保将 head 和 tail 对 MAX 取模时能够正确地计算出缓冲区中对应的索引号, 模数必须为 MAX 的倍数。MAX * 2 是满足这一要求的最小值, 同时也可保证当 MAX 为 2 的整数次幂时, MODULUS 也是 2 的整数次幂。代码中我们为 tail - head 的值加上 MODULUS, 其目的在于确保进行取模操作的值为正数, 但如果使用掩码操作来实现取模, 或者实现语言已经提供恰当的取模语义 (即与零相对应的值朝向 $-\infty$ 方向), 则额外加上 MODULUS 的操作可以省略。

如果可以使用特殊的值来表示缓冲区的空闲槽, 则环状缓冲区的实现可以得到进一步简化, 如算法 13.33 所示。

一种较为普遍的情况是缓冲区中仅有一个入队线程和一个出队线程 (例如 Oancea 等 [2009] 所使用的缓冲区), 此时环状缓冲区的实现可以更加简单, 如算法 13.34 所示。开发者需要意识到, 在不同平台上实现同一种算法时可能需要进行适当调整, 该算法即是一个典型案例。此处算法的工作流程与其在 Intel x86 处理器上的表现相同, 因为该处理器能够严格确保所有处理器以相同的顺序感知到存储操作的发生顺序。

但是在 PowerPC 平台上, 对于带 “\$” 标记的行, 则需要特别注意其执行顺序。一种策

略是插入内存屏障，如 Oancea 等所指出的那样。在 add 操作中，我们需要在 `buffer[tail]` 的写操作和 `tail` 的写操作之间（第 9 行之后）插入 `lwsync` 指令来作为存储-存储内存屏障（store-store memory fence）[⊖]。引入该指令之后，如果出队线程遵从适当的加载指令顺序，则必然会先感知到 `buffer` 的变化，然后再感知到 `tail` 的变化。同理，我们还需要在 `buffer[tail]` 的写操作之前（第 9 行之前）增加 `isync` 指令来作为加载-存储内存屏障（load-store memory fence）[⊖]，该指令可以确保处理器不会在加载 `head` 变量之前进行试探性的写操作，否则将可能导致 add 操作覆盖正在被出队线程读取的值。

类似地，在 `remove` 方法中，我们需要在加载 `buffer[head]` 和更新 `head` 之间（第 16 行之后）插入 `lwsync` 指令；同时需要在读取 `buffer` 之前（第 16 行之前）插入 `isync` 指令，该指令将在加载 `tail` 和从 `buffer[head]` 中加载数据之间扮演加载-加载内存屏障（load-load memory fence）的角色。

算法 13.31 基于细粒度锁的环状缓冲区

```

1  shared buffer[MAX]
2  shared head ← 0
3  shared tail ← 0
4  shared numAdded ← 0
5  shared numRemoved ← 0
6  shared addLock ← UNLOCKED
7  shared removeLock ← UNLOCKED
8
9  add(val):
10     lock(&addLock)
11     if numAdded - numRemoved = MAX
12         unlock(&addLock)
13         return false                                /* 入队失败 */
14     buffer[tail] ← val
15     tail ← (tail + 1) % MAX
16     numAdded ← numAdded + 1
17     unlock(&addLock)
18
19  remove():
20     lock(&removeLock)
21     if numAdded - numRemoved = 0
22         unlock(&removeLock)
23         return EMPTY                                /* 队列为空 */
24     val ← buffer[head]
25     head ← (head + 1) % MAX

```

⊖ `lwsync` 指令可以确保处理器在发射该指令之后，本处理器的所有后续指令必须等待 `lwsync` 指令之前的内存访问操作执行完毕且被所有其他处理器感知到。该指令的含义是“轻量级同步”（light-weight sync），同时也是 `sync` 指令的一种实现版本。与之相对应的，“重量级”（heavy-weight）同步的指令是 `sync`，该指令在普通的缓存内存之外还会处理输入/输出设备内存。`lwsync` 和 `sync` 指令在某种程度上都存在较大的开销，它们在实现上通常都要求处理器在进一步执行内存访问操作之前等待写缓冲区刷新完毕。这意味着处理器需要等待处理器间缓存同步的完成。

⊖ `isync` 指令可以确保处理器在发射该指令之后，本处理器的所有后续指令必须等待 `isync` 指令之前的指令执行完毕。该指令可以将某一点之前的内存加载操作与该点之后的内存访问操作隔离，但并不能确保其他处理器所感知到的指令执行顺序能够与本处理器的指令发射顺序保持一致（如果要达到这一要求，必须使用 `sync` 指令）。`isync` 指令可能更加高效的原因之一在于，其仅引入了处理器本地等待，即仅要求执行该指令的处理器等待其指令流水线为空；该指令本身不需要引入高速缓存一致性相关行为。


```

25  head  $\leftarrow$  (head + 1) % MAX
26  numRemoved  $\leftarrow$  numRemoved + 1
27  unlock(&removeLock)
28  return val

```

算法 13.32 所需变量较少的环状缓冲区

```

1  shared buffer[MAX]
2  MODULUS = MAX * 2                                /* 参见文中的解释 */
3  shared head  $\leftarrow$  0                             /*  $0 \leq \text{head} < \text{MODULUS}$  */
4  shared tail  $\leftarrow$  0                             /*  $0 \leq \text{head} < \text{MODULUS}$  */
5  shared addLock  $\leftarrow$  UNLOCKED
6  shared removeLock  $\leftarrow$  UNLOCKED
7
8  add(val):
9      lock(&addLock)
10     if (tail - head + MODULUS) % MODULUS = MAX
11         unlock(&addLock)
12         return false                                /* 入队失败 */
13     buffer[tail % MAX]  $\leftarrow$  val
14     tail  $\leftarrow$  (tail + 1) % MODULUS
15     unlock(&addLock)
16     return true                                    /* 入队成功 */
17
18  remove():
19      lock(&removeLock)
20      if (tail - head + MODULUS) % MODULUS = 0
21          unlock(&removeLock)
22          return EMPTY                                /* 队列为空 */
23      local val  $\leftarrow$  buffer[head % MAX]
24      head  $\leftarrow$  (head + 1) % MODULUS
25      unlock(&removeLock)
26      return val

```

算法 13.33 使用特殊值来表示空闲槽的环状缓冲区

```

1  shared buffer[MAX]  $\leftarrow$  [EMPTY,...]
2  shared head  $\leftarrow$  0
3  shared tail  $\leftarrow$  0
4  shared addLock  $\leftarrow$  UNLOCKED
5  shared removeLock  $\leftarrow$  UNLOCKED
6
7  add(val):
8      lock(&addLock)
9      if buffer[tail]  $\neq$  EMPTY
10         unlock(&addLock)
11         return false                                /* 入队失败 */
12     buffer[tail]  $\leftarrow$  val
13     tail  $\leftarrow$  (tail + 1) % MAX
14     unlock(&addLock)
15     return true                                    /* 入队成功 */
16
17  remove():
18      lock(&removeLock)
19      if buffer[head] = EMPTY
20          unlock(&removeLock)
21          return EMPTY                                /* 队列为空 */

```

```
22  val ← buffer[head]
23  head ← (head + 1) % MAX
24  unlock(&removeLock)
25  return val
```

算法 13.34 单入队线程 / 单出队线程无锁环状缓冲区 [Oancea 等, 2009]

```
1  shared buffer[MAX]
2  shared head ← 0          /* 下一个尝试出队的槽 */
3  shared tail ← 0          /* 下一个用于入队的槽 */
4
5  add(val):
6      newTail ← (tail + 1) % MAX
7      if newTail = head
8          return false
9      buffer[tail] ← val    $
10     tail ← newTail
11     return true
12
13 remove():
14     if head = tail
15         return EMPTY     /* 队列为空 */
16     value ← buffer[head]  $
17     head ← (head + 1) % MAX $
18     return value
```

Oancea 等还提出了另一种解决方案，即 `remove` 方法显式地将 `null` 作为 `EMPTY` 值写入槽中，而 `add`（或 `remove`）方法在将新值（或 `EMPTY` 值）写入槽之前必须先等待其所关注的槽变空（或者非空）。由于入队线程和出队线程各仅有一个，能够写入非空值和空值的线程也各仅有一个线程，且每个线程在执行写操作之前都会先观察另一个线程之前写入的值，所以两个线程对缓冲区的访问不会出现错误性的交错。类似地，`head` 和 `tail` 均只会会有一个线程操作，因而在最差情况下，某一线程最多只会获取已经过时的状态。该解决方案无需引入内存屏障，但是出队线程对缓冲区的操作可能会比基于内存屏障的策略引入更多的高速缓存冲突。Oancea 等将上述两种方案相结合，但是正如我们前面所论述的，任意一种方案均能独立满足要求。这些细节表明，在宽松内存顺序下正确地实现并发算法需要十分小心谨慎。

262
264

如果开发者是以缓冲区方式使用队列，即元素出队的顺序不必与其入队的顺序完全一致，则在该场景下实现无锁缓冲区并非难事。我们假定某个数组足够大，且永远不会发生回绕，算法 13.35 即为无锁缓冲区的一种实现方案，它假定缓冲区中所有槽的初始值均为 `EMPTY`。

该算法存在大量的重复扫描，为此，算法 13.36 引入了一个索引值 `lower` 来表示算法每次开始扫描的位置。该算法不仅需要识别出空槽，还需要能够识别出已被填充然后又被清空的槽，代码使用 `USED` 来表示此类槽的状态。

数组的长度不可能无限大，因而我们需要对上述算法进一步优化，即实现无锁环状缓冲区，如算法 13.37 所示。在此时的 `add` 方法中，在增加 `head` 的索引之前我们必须小心地将处于 `USED` 状态的槽修改为 `EMPTY`。此处索引值的范围也可与算法 13.32 类似，即其最大值可以达到 `MAX` 的 2 倍。

算法 13.35 基于数组的、无限长的无锁缓冲区

```

1  shared buffer[ ] ← [EMPTY,...]          /* 无限大的缓冲区 (无法实现) */
2  shared head ← 0                          /* 下一个入队槽 */
3
4  add(val):
5      pos ← FetchAndAdd(&head, 1)
6      buffer[pos] ← val
7
8  remove():
9      limit ← head
10     pos ← -1
11     loop
12         pos ← pos + 1
13         if pos = limit
14             return null                    /* 未在队列中找到元素 */
15     val ← LoadLinked(&buffer[pos])
16     if val ≠ EMPTY
17         if StoreConditionally(&buffer[pos], EMPTY)
18             return val

```

算法 13.36 基于数组的、无限长的无锁缓冲区 (增量扫描)

```

1  shared buffer[ ] ← [EMPTY,...]          /* 无限大的缓冲区 (无法实现) */
2  shared head ← 0                          /* 下一个入队槽 */
3  shared lower ← 0                         /* 每次扫描的起始索引值 */
4
5  add(val):
6      pos ← FetchAndAdd(&head, 1)
7      buffer[pos] ← val
8
9  remove():
10     limit ← head
11     currLower ← lower
12     pos ← currLower - 1
13     loop
14         pos ← pos + 1
15         if pos = limit
16             return null                    /* 未找到元素 */
17     val ← LoadLinked(&buffer[pos])
18     if val = EMPTY
19         continue
20     if val = USED
21         if pos = currLower
22             /* 尝试增加 lower */
23             currLower ← LoadLinked(&lower)
24             if pos = currLower
25                 StoreConditionally(&lower, pos+1)
26         continue
27     /* 尝试获取数据 */
28     if StoreConditionally(&buffer[pos], USED)
29         return val

```

算法 13.37 基于数组的有界无锁缓冲区

```

1  shared buffer[MAX] ← [EMPTY,...]
2  MODULUS = 2 * MAX
3  shared head ← 0                          /* 下一个用于填充的槽 */

```

```

4  shared lower ← 0                                /* 索引值从 lower 到 head-1 的槽可能会包含数据 */
5
6  add(val):
7      loop
8          currHead ← head
9          /* 在使用原子操作之前预读取数据 */
10         oldVal ← LoadLinked(&buffer[currHead % MAX])
11         if oldVal = USED
12             currLower ← lower
13             if (currHead % MAX) = (currLower % MAX)
14                 && (currHead ≠ currLower)
15                 advanceLower()                /* lower 已经落后 head 整个缓冲区的距离 */
16                 continue
17             /* 尝试将状态为 USED 的槽清空, 但前提是该槽中的数据未发生变化 */
18             if currHead = head
19                 StoreConditionally(&buffer[currHead % MAX], EMPTY)
20                 continue
21             if oldVal ≠ EMPTY
22                 if currHead ≠ head
23                     continue
24                 return false                    /* 状态已发生变化, 重试 */
25             currHead ← LoadLinked(&head)        /* 缓冲区满, 插入失败 */
26             /* 加载链接 / 条件存储内部的再次检测 */
27             if buffer[currHead % MAX] = EMPTY    /* 尝试占据 head 所表示的槽 */
28                 if StoreConditionally(&head, (currHead + 1) % MODULUS)
29                     buffer[currHead] ← val
30                     return true                    /* 插入缓冲区成功 */
31
32  remove():
33      advanceLower()
34      limit ← head
35      scan ← lower - 1
36      loop
37          scan ← (scan + 1) % MODULUS
38          if scan = limit
39              return null                        /* 未找到元素 */
40          /* 在使用原子操作之前首先预读取数据 */
41          val ← LoadLinked(&buffer[scan % MAX])
42          if val = EMPTY || val = USED
43              continue
44          /* 尝试将数据移出 */
45          if StoreConditionally(&buffer[scan % MAX], USED)
46              /* 注意: 由于此处并不是队列, 所以移出并非 USED 或者 EMPTY 的元素通常都是安全的 */
47              return val
48  advanceLower():
49      if buffer[lower % MAX] ≠ USED
50          return                                /* 在使用原子操作之前快速返回 */
51      loop
52          currLower ← LoadLinked(&lower)
53          if buffer[currLower % MAX] = USED
54              if StoreConditionally(&lower, (lower + 1) % MODULUS)
55                  continue
56          return

```

13.8.4 支持工作窃取的并发双端队列

为支持工作窃取, Arora 等 [1998] 设计了一种无锁双端队列。本地工作线程可以在队列中压入或者弹出工作单元, 同时其他线程也可从中移出 (窃取) 工作单元。在该设计中, 本

地工作线程从双端队列的一端执行压入或者弹出操作，而其他线程则从另一端完成工作窃取（即：双端队列只有一个输入端）。算法 13.38 展示了基于加载链接 / 条件存储的实现方案^①，该方案可以避免 ABA 问题。我们也可为 tail 端的索引绑定一个计数器，从而可以基于 CompareAndSwap 安全地实现相同的目的。

对于本地工作线程而言，向双端队列中压入数据十分简单，且无需任何同步操作，而弹出操作则需要判断所弹出的元素是否为双端队列中的最后一个，如果结果为真，则可能和其他的非本地窃取线程产生竞争。本地工作线程和其他线程都可能会尝试更新 tail，只有竞争的优胜者才能真正获取双端队列中的工作单元。在产生竞争的情况下，不论本地工作线程是否竞争成功，其都会将 tail 设置为零（第 26 行），但这并不会影响其他正在进行工作窃取的线程，因为在出现竞争时，其他线程要么会竞争失败（从而无法获取工作单元），要么已经窃取成功（此时对 tail 的修改将不影响任何线程）。另外需要特别注意的是，pop 方法将 top 置零的操作必须先于将 tail 置零的操作，从而确保 remove 过程永远满足 $top \leq tail$ 这一不变式。使用 top 和 tail 两个变量还存在另一个优势，即通过 $top - tail$ 便可快速得出双端队列中元素的个数（除非是在将这两个变量都置为零的过程中，此时它们之间的差可能为负数）。

算法 13.38 支持工作窃取的无锁双端队列 [Arora 等, 1998]

```

1  shared deque[MAX]
2  shared top ← 0                      /* 双端队列中最后入队元素的索引值 + 1 */
3  shared tail ← 0                     /* 双端队列中最先入队元素的索引值 */
4
5  push(val):                          /* 本地工作线程通过该方法来压入工作单元 */
6      currTop ← top
7      if currTop ≥ MAX
8          return false                /* 双端队列溢出 */
9      deque[currTop] ← val
10     top ← currTop + 1
11     return true                      /* 压入成功 */
12
13 pop():                              /* 本地工作线程从双端队列的本地端弹出一个工作单元 */
14     currTop ← top - 1
15     if currTop < 0
16         return null                 /* 双端队列为空 */
17     top ← currTop
18     val ← deque[currTop]
19     currTail ← LoadLinked(&tail)
20     if currTop > currTail
21         return val                  /* 此时不会与其他线程发生竞争 */
22     /* 可能与其他线程产生竞争，队列可能为空 */
23     top ← 0
24     if StoreConditionally(&tail, 0)
25         return val                  /* 本地工作线程竞争胜出，并成功获取一个工作单元 */
26     tail ← 0
27     return null
28
29 remove():                           /* 从其他线程的双端队列中窃取工作单元 */
30     loop

```

① 该算法中的变量名称与 Arora 等 [1998] 文献中的稍有不同。在该算法中，我们将队列中由本地工作线程访问的一端的索引称为 top，而将另一端的索引称为 tail，此时对于本地工作线程而言，双端队列表现得更像是一个栈，而对于其他线程而言，双端队列则更像是队列。


```
31 currTail ← LoadLinked(&tail)
32 currTop ← top
33 if currTop ≤ currTail
34     return null /* 双端队列为空 */
35 val ← deque[currTail]
36 if StoreConditionally(&tail, currTail+1)
37     return val /* 在设置 tail 的竞争中胜出, 成功获取一个工作单元 */
38 /* 与其他窃取线程产生竞争, 或者 pop 操作已导致双端队列为空 */
39 /* 如果工作窃取并非必须, 则此处可以返回失败而非继续循环 */
```

13.9 事务内存

在介绍事务内存 (transactional memory) 与垃圾回收之间的关系之前, 我们有必要先介绍事务内存。

13.9.1 何谓事务内存

所谓事务 (transaction), 是指一组读写操作集合必须“看起来”原子化地执行, 从表现形式来看, 组成事务的读写操作之间不应当穿插其他读写操作。LoadLinked/StoreConditionally 实现了针对单个字的事务语义, 但此处我们所关注的是针对多个独立字的事务操作。合理的事务机制一般包括如下几个单元:

- 事务的开始 (start)。
- 当前事务中的读 (read) 操作。
- 当前事务中的写 (write) 操作。
- 事务的结束 (end)。

事务的结束通常称为 (尝试) 提交 (commit)。如果成功, 则事务生效, 否则事务的所有写操作都将被抛弃, 此时软件可以进行重试, 也可采取其他一些操作。因此事务可以“投机性” (speculatively) 地执行。事务的结束必须是显式的, 只有这样投机操作才会有定论, 事务才可能被受理, 写操作才可能生效或被拒绝, 软件才能据此进行重试或者采取其他操作。

数据库领域中的事务必须满足 ACID 要求, 类似地, 事务内存也必须满足如下要求:

- 原子性 (atomicity): 事务的全部操作要么全部成功, 要么就像没有发生一样。
- 一致性 (consistency): 事务的执行应当看起来是在一个瞬间完成的。
- 隔离性 (isolation): 任何其他线程都无法感知到事务的中间状态, 而只能感知到事务执行之前和执行之后的状态。

持久性 (durability), 即数据库中事务的最后一个重要特征, 它不允许成功执行的事务发生丢失 (或者要保证极高的可靠性), 但这一要求并不适用于事务内存。

事务执行过程中真正的读写操作可能会散布在多个时间点, 因而如果多个事务在执行过程中访问相同的内存地址, 则它们的读写操作之间可能会产生干扰。例如, 对于 A 和 B 两个事务, 如果事务 A 在写某个值的同时, 事务 B 正在读取或者修改该值, 则会产生冲突。存在冲突的事务必须遵从一定的顺序。某些情况下, 我们不可能令事务中的读写操作在事务提交成功之前得到真正执行。例如, 如果事务 A 和 B 同时读取变量 x, 然后同时尝试修改该变量, 则根据事务的语义, 两个事务不可能都执行成功。在这种情况下, 至少一个事务需要中止 (abort), 并确保该变量的状态与被中止的事务根本没有发生时的状态一致。一般情况下, 软件会发起重试, 且重试通常会对事务的执行顺序进行强制要求。

265
267

事务内存可以基于硬件实现,也可基于软件实现,还可使用软硬件混合方式实现。任何一种实现策略都必须提供以下几种操作:原子化写操作、冲突检测、可见性控制(visibility control)(以支持隔离性)。可见性控制可以作为冲突检测的一部分。

写操作的原子化可以通过缓冲(buffer)或者撤销(undo)的策略实现。缓冲策略会先在内存中的某临时位置执行写操作,并且仅在事务提交时才将最终的值写入指定位置。硬件缓冲策略可以通过增加缓存或者其他额外缓冲区的方式实现,而软件缓冲则可能会借助于与待修改值同级别的字/域/对象。引入缓冲之后,事务的提交要么会将缓冲的写操作生效,要么直接将缓冲区抛弃。大多数情况下,事务的成功提交通常需要较多工作,相比之下中止事务的开销相对较小。基于撤销的策略则与缓冲策略截然不同,其事务执行过程中的写操作会直接修改数据,但每个写操作会在数据修改之前将原有的值记录到撤销日志(undo log)中。如果事务最终成功提交,则可以直接抛弃执行过程中的撤销日志,而一旦事务被中止,则必须使用撤销日志来恢复被修改的值。与缓冲策略类似,撤销日志也可使用硬件、软件、软硬件混合的方式实现。

冲突检测既可以是积极(eagerly)的,也可懒惰(lazily)的。积极的冲突检测策略会在每次内存访问之前检测该操作是否会与当前正在执行的事务产生冲突,如果发现两个事务可能发生冲突,那么在必要情况下可以将其中的一个中止。懒惰冲突检测策略则只有在尝试提交事务时才进行冲突检测。某些冲突检测机制还允许事务在执行过程中检测当前是否有冲突发生。软件实现策略可以设置对象头部的旗标,或者使用额外的表来记录对象的访问,事务性访问在冲突检测过程中会对其进行检测。类似地,硬件实现策略则通常会将旗标与高速缓存行或者被修改的字相关联。

为简化表述,我们将介绍一种简单的基于硬件的事务内存接口,该接口是由如下几种原语构成的。此处的描述方式与 Herlihy 和 Moss [1993] 文献中的相同。

- TStart() 表示事务的开始。
- TCommit() 表示事务尝试进行提交,该原语返回一个布尔值,如果为真,则表示事务提交成功。
- TAbort() 表示事务尝试中止执行,由程序主动发起中止请求在某些情况下比较有用。
- TLoad(addr) 表示对指定地址发起事务性的加载操作。该操作会将目标地址添加到事务的读集合中,同时返回该地址的当前值。
- TStore(addr, value) 表示事务性地将指定的值写入目标地址。该操作会将目标地址添加到事务的写集合中,并事务性的执行写操作,即:如果事务被中止,则该写操作不会产生任何作用。

许多并发数据结构可以基于上述几种原语进行简化,例如算法 13.39 便是对算法 13.30 的简化。引入事务内存之后,由于我们可以原子化地对两个不同地址的值进行更新,因而 add 函数可以得到简化;类似地,由于可以原子性地读取两个甚至三个不同的值,remove 操作也可得到简化。更重要的是,我们可以更加清晰地判断出事务的实现是否正确,而如果要简化之前的算法进行正确性分析,则必须对读写操作的执行顺序进行更加谨慎的论证。

算法 13.39 基于事务内存的单链表并发队列

```

1  shared head ← new Node(value: dontCare, next: null)
2  shared tail ← head
3

```

```

5    node ← new Node(value: val, next: null)
6    loop
7        currTail ← TLoad(&tail)
8        TStore(&currTail.next, node)
9        TStore(&tail, node)
10       if TCommit()
11           return
12
13 remove():
14     loop
15         currHead ← TLoad(&head)
16         next ← TLoad(&currHead.next)
17         if next = null
18             if TCommit()          /* 提交操作可以确保执行过程中变量的一致性 */
19                 return EMPTY      /* 队列为空 */
20             continue
21
22         /* 队列非空, 尝试移除其首节点 */
23         val ← TLoad(&next.value)
24         TStore(&head, next)
25         if TCommit()
26             return val
27         /* 移除失败, 进行重试 */

```

13.9.2 使用事务内存助力垃圾回收器的实现

事务内存与垃圾回收之间的关系主要表现在两个方面。一方面, 事务内存可以作为垃圾回收器的实现技术之一 [McGachey 等, 2008]; 另一方面, 事务可能会是托管语言的基本语义之一, 回收器必须能够正确地支持这一语义。本小节我们将介绍事务内存垃圾回收器实现过程中的应用, 而下一小节我们将介绍垃圾回收器对托管语言中事务语义的支持。

毋庸置疑, 事务内存可以简化并发数据结构的实现, 因而其可以简化并行 / 并发分配 / 回收的实现, 特别是并发分配器、赋值器、回收器、读写屏障以及并发回收器数据结构的实现。目前事务内存尚不存在统一的硬件实现标准, 同时在软件方面也存在多种不同的实现方式, 因而我们很难指定某种标准并进行描述, 但不论如何, 在自动内存管理中使用事务内存需要注意以下几个方面:

- 软件事务内存 (software transactional memory) 通常会引入显著开销, 即使在经过优化之后。如果要求自动内存管理系统中大部分组件的运行时开销都足够小, 则软件事务内存的应用场景可能会受到相当大的限制。另外, 访问频率不高的数据结构也可以使用锁来降低实现复杂度。
- 硬件事务内存 (hardware transactional memory) 通常会包含一些与硬件相关的特殊要求。例如, 冲突检测、访问与更新均必须以物理单元为最小粒度来执行 (例如高速缓存行)。由于硬件容量通常存在限制 (例如组相关高速缓存中每个缓存集合所包含的缓存行数量), 所以某些硬件事务内存的实现可能会对事务中所涉及的数据有总量限制。另外, 开发者所设计的变量布局与其在高速缓存行上的布局可能存在差异, 因而开发者还需额外注意一些底层的实现细节。
- 事务内存存在大多数情况下可以轻易满足无锁要求。即使事务内存的底层提交机制满足无等待要求, 事务之间依然可能发生冲突, 进而导致事务的中止或者重试。无等待数据结构的开发依然十分复杂, 且需要注意许多细节。

- 事务内存需要开发者仔细进行性能调整。需要关注的内容之一是多个事务访问相同数据结构时的固有冲突。例如对于并发栈而言，其瓶颈在于不同线程更新栈顶指针的操作，但事务内存并不能解决这一问题。另外，对事务中的读写操作进行任何方式的调整（例如将其移动到靠近事务开始或者结束的位置）都会显著影响冲突发生的概率，以及事务的重试开销。

271

综上所述，硬件事务内存设施十分有用。例如基于 Advanced Micro Devices 设计的先进的同步设备 [Christie 等, 2010]，其接口与我们上一小节所描述的接口十分类似，该方案支持在一个事务中对至少四个完全独立的高速缓存行进行读写操作，这足以简化本章绝大部分并发数据结构的实现。但是，基于硬件事务内存的简化算法是否会带来性能上的改进，目前仍尚无定论，但毋庸置疑的是，基于事务内存的简单模型可以减少错误，并且降低开发难度。

13.9.3 垃圾回收机制对事务内存的支持

接下来，我们将探讨事务内存与垃圾回收的另一种关系，即编程语言如何同时支持自动内存管理以及某些内建的事务内存语义 [Harris and Fraser, 2003 ; Welc 等, 2004, 2005]。此处的核心问题在于，垃圾回收和事务内存这两种机制的执行可能会相互干扰，特别是在并发程度很高的场景中。

可能存在的一种干扰形态是：内存管理器可能导致事务产生冲突，进而增大事务的重试开销，从而影响到赋值器或者回收器的正常执行，或者同时影响两者。例如，如果赋值器尝试发起一个执行时间较长的事务，而其执行过程又与回收器产生冲突，则赋值器事务可能会持续性地被回收器中止，或者回收器可能会被阻塞较长的一段时间。如果事务本身使用硬件事务内存来实现，则情况将更加复杂。例如，并发回收器针对某一对象所进行的标记、转发、复制等操作均有可能中止赋值器事务的执行，而原因仅仅是回收器与执行中的事务访问了相同的内存——即使语言的实现者已经通过精心设计来确保它们之间不会相互干扰，但这一问题仍不能避免。由于硬件不可能意识到其所操纵的数据为托管数据，所以硬件事务内存无法解决这一问题，相比之下，针对特定语言设计的软件事务内存则可能会将对象头部与其数据域区分对待。

事务的执行还可能受到内存回收语义的影响。例如，假设某一事务内存系统使用原地更新的实现策略，同时使用回滚日志来记录被修改数据原有的值，并据此来支持事务的正常中止。这一场景下，可能会出现某一对象仅从回滚日志可达的情况，此时如果事务处于未决状态，则该对象不应当被回收器判定为不可达。因此，回收器还需要把事务日志当作根集合的一部分来处理。另外，对于复制式回收算法，回收器不仅需要对回滚日志中的指针进行追踪，而且还必须将其更新到其目标对象的新地址。

事务性语言中的内存分配是一个值得关注的问题。如果事务在执行过程中存在对象分配，但事务最终被中止，则从逻辑上讲，所分配的对象应当以某种方式回收。但是，一旦分配操作会涉及全局共享数据结构，如果我们要在事务中止时精确恢复到事务执行之前的状态，则意味着事务需要对空闲链表或者阶跃指针加锁，直到事务提交或者中止，这显然是无法接受的。因此，分配过程的回滚应当更加偏向于逻辑上而非物理上的操作。例如对于基于空闲链表的系统，事务在被中止时应当将其所分配的对象释放，这一分配释放过程可能会导致空闲链表中的节点的位置发生变化，也可能导致内存块分裂（且无法合并）。编程语言也

有可能允许事务在执行过程中发起一些非事务性的操作，这些操作所分配的对象可能会暴露给其他线程，因而它们不应当随着事务的中止而释放，同时这些对象初始化过程中所执行的操作（例如设置对象头域）也不应当回滚。诸如 Open Nesting[Ni 等, 2007] 等概念可能会对这一场景的处理有所帮助，一个通用的解决方案是将事务性赋值器操作中的所有原子化内存管理操作都当作 open nested 操作。

272

最后，某些事务内存系统会在执行过程中产生大量的内存分配操作，进而增加分配器与回收器的负担。特别是在某些软件事务内存的实现中，事务在修改某一对象之前必须先创建该对象的副本，并基于副本进行修改，同时只有在事务成功提交的情况下才使用副本取代原有对象。在这一情况下，分配器在语义上并无更多需要处理的内容，但其负载可能会发生变化。事务内存和垃圾回收之间的一个共同点在于，它们都需要高效的、满足合适前进保障要求的并发数据结构。例如，事务的提交便是一致性算法的一个应用场景，理想情况下其应当满足无等待要求。

13.10 需要考虑的问题

我们需要首先明确的是，正确地实现并发算法非常困难！这一结论并不夸张，因此除非万不得已，最好避免使用并发算法。但是，在现代硬件条件下，并发编程确实变得越来越必要，因而我们才会专门以一章的篇幅来对其进行描述。

系统需要支持哪些平台？目标平台的内存一致性特征如何？其提供哪些内存屏障与同步原语？我们应当尽量降低算法对执行顺序的依赖，但在某些平台上，我们仍可能需要在算法中插入屏障或者其他原语，例如 13.8 节中的算法 13.34。哪些执行顺序需要引入屏障？

平台支持哪些原子更新原语？尽管 LoadLinked/StoreConditionally 原语便于使用且更加高效，但大多数流行系统仅支持 CompareAndSwap 或者等价原语，后者可能会遇到 ABA 问题，但该问题也可通过某种方式来避免，如算法 13.27 所示。或许在不久的将来，事务内存将逐渐实用化。

算法需要达到何种级别的前进保障要求？较弱的前进保障更容易实现，也更易于推导。对于访问量较低的数据结构，直接进行加锁可能会更加合适，因为与具有更强前进保障级别的无锁算法相比，加锁不仅容易实现，而且更容易正确地实现。另外，即使在某些已发布的、绝大多数场景满足无等待要求的系统中，某些极端情况的处理仍会使用更加简单的实现技术，将这些非关键场景无等待化并不具有太大价值。

系统是需要实现真正的并发（即允许多个线程在硬件上同时执行），还是仅需要达到多程序（multiprogrammed）级别？多程序并发算法的实现通常会更加容易。

在后续章节中，我们将以本章所介绍的内容为基础来描述并行、增量、并发以及实时回收器。

273

并行垃圾回收

现代硬件体系架构的发展趋势是处理器以及处理器核心数量将越来越多。Sutter[2005]曾经指出，使用传统方法来提升性能的“免费午餐”已经吃尽。能耗以及散热问题导致硬件架构无法进一步提升时钟频率（时钟频率的提升与其功耗的增长为三次方关系），因而硬件设计者转向在单个芯片上集成多个处理器核心（核心数量的增长与其能耗的增长呈线性关系）。目前并无证据表明处理器的这一发展趋势会发生变化，因而在应用程序的设计与实现中，充分利用硬件的并行能力来提升性能将变得越来越重要。而在另一方面，异构（heterogeneous）与非一致（non-uniform）内存架构只会增加开发者的负担，因为开发者需要仔细考虑底层平台的各种特殊性质。

到目前为止，我们所介绍的算法均假定存在多个赋值器线程，但仅存在一个回收器线程，对于现代多核或多处理器而言，这将是极大的资源浪费。本章我们将考虑如何将垃圾回收并行化，但我们依然假设在垃圾回收处理过程中，所有赋值器线程均被挂起，同时只有在垃圾回收完成之后，赋值器线程才能继续执行。一些早期的论文可能会将“并发”（concurrent）、“并行”（parallel）、“即时”（on-the-fly）、“实时”（real-time）这几个术语等价或者混淆，但在本书中，我们将更加注重这些术语的前后一致性，并与当前业界的普遍用法保持一致。

图 14.1a 以水平条带来表示单个处理器的执行，时间的前进方向为从左到右，其中白色部分表示赋值器的执行，其他颜色表示回收器的执行。灰色方框表示一次垃圾回收操作，而黑色方框表示下一次回收操作。在多处理器环境下，挂起赋值器意味着所有赋值器线程将陷入停顿。图 14.1b 展示了我们到目前为止所描述过的一般回收情形，即回收器将多个赋值器线程挂起，并使用一个处理器来完成垃圾回收工作，此时硬件资源显然得不到充分利用。毋庸置疑，使用多处理器共同完成垃圾回收工作将减少停顿时间（赋值器线程同样需要全部挂起），如图 14.1c 所示，这便是本章将要介绍的并行回收（parallel collection）。

对于所有需要将赋值器线程全部挂起，直到回收结束的回收策略，我们将其称为万物静止式回收（stop-the-world collection）。我们曾经提到，降低回收停顿时间的策略还包括允许赋值器和回收器交替执行的增量回收（incremental collection），以及允许赋值器线程和回收器线程同时执行的并发回收（concurrent collection），我们将在稍后的章

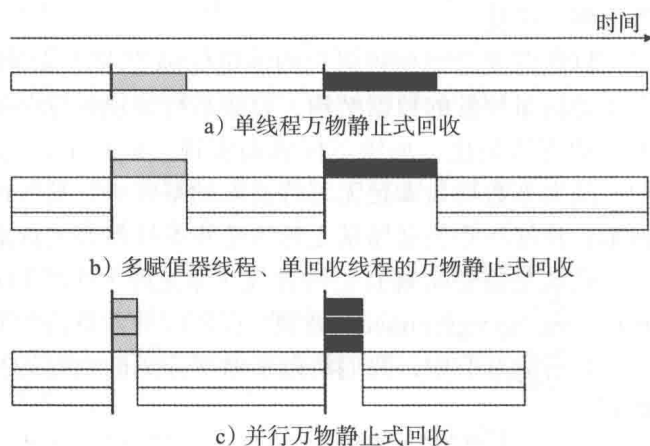


图 14.1 万物静止式垃圾回收。每个条带表示一个处理器的执行过程。不同颜色（非白色）的区域表示不同的回收周期

节中描述这些回收策略。本章将专注于并行追踪式垃圾回收算法。第 5 章曾经提到，引用技术算法天然就具有并行和并发特性，我们将在第 18 章讨论如何提升多处理器环境下引用计数算法的性能。本章将讨论如何将并行技术应用于追踪式垃圾回收器的 4 个主要组成部分，即：标记、清扫、复制、整理。

14.1 是否有足够多的工作可以并行

并行回收的目的在于充分利用硬件资源来降低垃圾回收的时间开销。对于万物静止式回收而言，并行回收可以降低停顿时间；而对于增量回收或者并发回收而言，并行回收有助于缩短一个回收周期的时间。对于任何并行问题，首先必须考虑是否有足够多的工作值得使用并行方式。并行回收不可避免地会在各回收线程之间引入一些同步操作，从而增大了回收开销。某些算法可能需要使用锁，而其他算法可能需要使用诸如 CompareAndSwap 的原子操作原语，并需要对辅助数据结构进行精心设计。但不论将同步机制优化到何种程度，它们都不可能达到单处理器解决方案的执行效率。因此并行回收首先需要考虑的问题便是：是否有足够多的垃圾回收工作可以并行，且并行解决方案所带来的收益是否可以超出其所引入的额外开销。

某些垃圾回收问题的出现可能不利于使用并行回收。例如当标记 - 清扫回收器对链表进行追踪时，链表固有的顺序特征决定了在追踪阶段的每一步中，标记栈中都仅会包含一个元素，即链表中下一个将被追踪的节点。在这一场景下，只有一个回收线程处于工作状态，其他线程均处于等待状态。Siebert [2008] 给出了在包含 p 个处理器的系统中，由于缺乏工作而处于等待状态的标记线程数量 n 与任意可达对象 o 的最大深度之间的关系：

$$n \leq (p-1) \cdot \max_{o \in \text{reachable}} \text{depth}(o)$$

该公式假设标记阶段所有操作花费的时间均相同，但这往往是不切实际的，扫描所需的时间往往取决于被扫描对象的类别（kind）。尽管在大多数编程语言中，绝大多数对象通常都相对较小，即它们仅包含数量较少的指针，但数组却可能会很大，且通常会比一般对象大得多（除非使用串联子数组的方式实现数组，即数组是由一系列固定大小的子数组构成）。

幸运的是，应用程序所使用的数据结构往往都不会是单链表，而是包含多种不同的数据结构。例如，当对具有分支结构的数据结构（例如树形结构）进行追踪时，在触达叶节点之前，追踪阶段的每一步所产生的工作量都会比其所处理的工作要多。另外，回收器通常还可以从多个位置发起追踪过程，包括全局变量、赋值器线程栈、分代回收器与并发回收器中的记忆集等。Sibert 对小型 Java 基准程序进行研究发现，不仅许多应用程序中可达对象的最大深度都非常浅，而且更重要的是，可达对象的最大深度与可达对象数量之间的比值往往都非常小，即不足 4%，这表示并行追踪可以达到很高的并行程度：所有基准程序在处理器数量达到 32 个时均表现良好（某些场景下甚至可以支持更多处理器）。

追踪过程是整个垃圾回收过程中最难以并行化的部分。对清扫过程或者整理式回收中修正引用等过程的并行化则要直接得多，至少在原则上如此。一种显而易见的策略是将堆划分为多个不重叠的区域，每个处理器负责一个区域的处理，具体的实现细节决定了并行处理算法的成败。

14.2 负载均衡

并行解决方案应当满足的第二个要求是：回收工作在可用硬件资源上的分配方式应当尽

量减少对同步操作的依赖，从而尽量将所有处理器保持在忙碌状态。缺乏负载均衡的简单并行算法很可能无法提升多处理器环境下的回收速度 [Endo 等, 1997]。但不幸的是，负载均衡化和同步开销最小化这两个目标通常存在冲突。静态负载均衡 (static load balancing) 策略的工作分配在处理过程开始之前就已经确定，即在内存管理器启动时，或者至少是在第一次垃圾回收之前。该策略中，各回收线程只需要在回收工作何时完成这一问题上达成一致即可，除此之外，它们之间不需要再引入任何同步。但是，静态负载均衡策略通常无法确保各线程的工作分配均匀。例如，工作在 N 处理器环境下、使用大块连续内存的并行标记-整理策略可能会将堆空间划分为 N 个区域，每个处理器负责一个区域内的引用修正工作。该策略的实现相对简单，但每个处理器的回收工作量却取决于它们所负责区域内的对象数量，以及这些对象所包含引用的数量。除非所有区域的对象特征大体一致，否则某些处理器的工作量必然会高于其他处理器。需要注意的是，处理器之间需要进行平衡的不仅仅是工作量，对其他资源的分配进行平衡也十分重要。在 Halstead [1984, 1985] 所实现的 Baker 式 [1978] 并行复制回收器中，每个处理器都拥有专属的来源空间与目标空间，但不幸的是，在这一静态组织方式下经常会出现某一处理器耗尽其目标空间而其他处理器的目标空间却存在富余的情况。

[277]

许多回收任务需要通过动态负载均衡 (dynamic load balancing) 策略来确保工作分配的近似均衡。对于在执行之前可以预估出工作量的任务，即使不同回收周期内的预估结果可能存在差异，线程之间的工作划分仍可以较为简单地实现，同时各并行回收线程在后续执行过程中也无需引入其他同步。例如，在并行标记-整理回收器的整理阶段，Flood 等 [2001] 依照已经识别出的存活对象将堆空间划分为 N 个区域，并确保每个区域中存活对象的总内存量近似相等，每个处理器可以独立对一个区域进行并行整理。

但是在更多情况下，我们通常很难对所需执行的工作进行事先预估并据此进行划分。为此更加一般化的解决方案是将所需执行的工作划分为比回收线程 (处理器) 数量更多的子任务，同时驱使线程一次获取一个子任务来执行。这种细粒度划分 (over-partition) 策略存在诸多优势：较小的子任务更容易适应处理器数量不同的硬件平台，从而提升了并行回收的弹性；如果某一子任务的实际处理时间比预期的时间要长，则剩余工作可以由其他已完成较小工作任务的线程承担。例如，Flood 等在设置转发指针之前会以对象为单位将堆划分成 M 个大致相等的区域， M 通常是回收线程数量的 4 倍。每个线程负责一个区域的处理，该过程同时会统计存活对象的总内存量，并将相邻未标记对象合并成单个垃圾内存块。需要注意的是，该回收器会在不同的回收阶段使用不同的负载均衡策略 (我们将在稍后进行更加详细的介绍)。

我们将本章后续部分将要介绍的每种算法浓缩为 3 个主要的子任务，即获取工作、执行工作、生成新工作。我们假定在大多数情况下，每个回收线程均依照如下的抽象方式执行：

```
while not terminated()
    acquireWork()
    performWork()
    generateWork()
```

该抽象算法中，acquireWork 尝试获取一个或者多个工作单元，performWork 执行工作单元，generateWork 将 performWork 在执行过程中新发现的一个或者多个新工作单元添加到全局任务池中，以便各回收线程获取。

14.3 同步

可能会有读者认为，最佳的负载均衡方式是将回收工作划分为尽量小的工作单元，例如对单个对象进行标记。如此细粒度的划分方式可以完美平衡各处理器之间的负载（因为此时任意一个工作单元都可以被任意一个申请获取任务的处理器执行），但在该策略下处理器之间的同步开销将无法接受。处理器之间进行同步的目的有二：一是确保回收过程的正确性，二是最大限度地减少重复工作。正确性包括两方面的含义：一方面要避免并行执行的回收线程破坏堆，另一方面要避免其破坏回收器自身数据结构。例如，任何移动式回收器都必须保证一个对象只可能被一个线程所复制，如果两个线程同时复制同一对象，则在最乐观的情况下（即对象不可修改）仅会造成空间上的浪费，而在最差情况下两个副本将出现不一致。对回收器自身数据结构进行保护也是十分必要的，如果所有线程共享同一个标记栈，则所有的压入与弹出操作都必须是同步的，只有这样才能避免多个线程同时操作栈或者增加 / 移除元素时可能出现的工作丢失问题。

各回收线程之间的同步存在时间和空间两方面的开销。线程在进行独占式访问时可能会使用锁或者无等待数据结构，因此设计良好的算法应当尽量减少需要进行同步的操作，例如使用线程本地数据结构。对于需要进行同步的操作，应当确保其在大多数情况下能够执行到速度较快的分支，例如锁竞争的情况应当极少发生，或者诸如 `CompareAndSwap` 等原子操作应当在绝大多数情况下都执行成功。如果竞争失败，则使用无等待方式竞争其他工作通常要比重试策略要好。但在某些情况下，即使不使用独占式访问，回收的正确性也能得到保证，此时便可省略同步操作。例如设置对象头部的标记位即是一个幂等操作，两个线程设置相同标记位的唯一风险在于回收器执行了一些不必要的工作，但其开销通常会比执行同步操作要小得多。

[278]

并行回收的具体实现需要在负载均衡和同步开销方面做出平衡。现代并行回收器通常会令各回收线程竞争较大的工作任务，同时尽量避免线程在处理任务的过程中引入其他同步操作。工作任务有多种组织方式：线程本地标记栈、堆中不同的扫描区域、（固定大小的）工作缓冲区池等。当然，这些数据结构往往也会拥有其自身的元数据，并可能引入一定的碎片，因而也会产生一定的空间开销，但这一开销通常较小。

14.4 并行回收的分类

在本章的后续部分，我们将介绍并行标记、并行清扫、并行复制、并行整理的具体解决方案，所有算法均假定在回收线程从开始执行到执行完毕的过程中，赋值器线程均挂在在安全回收点。我们尽可能在一个统一的框架内对各种场景进行研究。不论是对于哪种情况，我们均需要关注算法如何实现回收工作的获取（`acquire`）、执行（`perform`）、生成（`generate`），这3种操作的设计与实现决定了算法所需同步操作的类型、单个回收线程的负载粒度，以及在多个处理器之间进行负载均衡的策略。

并行垃圾回收算法大致可以划分为两大类，即以处理器为中心（`processor-centric`）的并行算法和以内存为中心（`memory-centric`）的并行算法。在以处理器为中心的算法中，线程所获取的工作通常大小不等，且线程之间通常会存在工作窃取行为。该策略通常不关注线程所处理对象的位置，但通过前面的章节我们知道，即使是在单处理器环境下，局部性都会对处理性能产生显著影响，而对于非一致内存或者异构系统而言，局部性的作用则更加重要。

以内存为中心的算法会更多考虑局部性因素，此类算法通常会针对堆中连续内存块进行操作，并从共享工作缓冲区池中获取工作包（或者将工作包释放到全局工作池中），工作包的大小通常固定。绝大多数并行复制式回收器均使用这一策略。

最后让我们关注并行回收的结束。回收线程不仅会尝试获取工作，而且还会进一步动态地生成新的工作。因此，简单判定共享工作池是否为空通常不足以断定回收过程是否结束，因为活动线程可能还会向工作池中加入新的任务。

14.5 并行标记

[279]

标记过程包含 3 种操作：从工作列表中获取一个对象、检测并设置标记位、将对象的子节点添加到工作列表以确保标记工作的持续。到目前为止，所有已知的并行标记算法都是以处理器为中心的。如果使用线程本地工作列表，则在工作列表不为空的情况下，从中获取对象无需任何同步操作，否则线程必须原子化地获取新工作（即一个或多个对象），可以从其他线程的工作列表中窃取，也可从全局工作列表中获取。原子化的目的主要是为了确保线程在从其他工作列表中获取工作时不破坏其完整性。对于非移动式回收器，多次标记同一对象或者多次将其添加到工作列表仅会影响回收的性能，但不会对回收的正确性造成影响。尽管在最差情况下，某一线程可能会在其他线程处理完某一数据结构之后再次对其进行处理，但在实践中这一情况通常很少发生。因此，如果使用对象头部的一个位或者字节图中的一个字节来记录对象是否已得到标记，则标记位的检测与设置均可以通过非原子化的读写操作完成。但是，如果使用位图中的位来记录对象是否已得到标记，则在设置标记位时必须使用原子操作。如果工作列表为线程私有，且容量无限，则将对象子节点添加到工作列表中也无需使用同步操作。反之，如果工作列表是共享的，或者其容量有限，则同步操作将不可避免。在工作列表容量有限的情况下，一旦线程本地工作列表被填满，则其必须将部分工作释放到全局工作列表中。如果对象是非常大的指针数组，则一次性将其子节点全部添加到工作列表中可能会影响线程之间的负载均衡。某些回收器（特别是应用于实时系统的回收器）会增量式地处理大对象的指针域，其实现方法通常是将大对象通过链式数据结构实现，而非一大块连续数组。

以处理器为中心的并行回收

工作窃取（work stealing）。Endo 等 [1997]、Flood 等 [2001]、Siebert [2010] 均使用工作窃取策略来实现负载均衡，即一旦某线程完成标记任务，其将从其他线程的工作列表中窃取工作任务。在 Endo 等的并行 Boehm and Weiser [1988] 保守式标记 - 清扫回收器中，他们为每个线程都配备了本地标记栈以及可窃取工作队列（stealable work queue）（见算法 14.1）。每个线程会周期性地检查其可窃取标记队列是否为空，若检查结果为空，则将其标记栈中的所有工作转移到该队列中（本地根除外）。处于空闲状态的线程将首先检查其本地可窃取队列，若其为空，则将进一步检查其他线程的可窃取队列。一旦其发现了非空闲队列，其会将队列中的一半元素“窃取”到本地标记栈中。多个线程可能会同时尝试工作窃取，因此可窃取队列必须通过锁来保护。Endo 等人发现，如果使用加锁 - 窃取的实现策略，处理器会将大量时间耗费在加锁过程中，因此他们将工作窃取的策略修改为：尝试加锁，如果成功，则执行窃取，否则，便将其跳过。也就是说，如果线程发现某一队列已被加锁，或者尝试加锁失败，则将跳过该队列并尝试从下一个队列中进行窃取。这一执行序列满足无锁要求。

算法 14.1 Endo 等人 [1997] 的并行标记 – 清扫算法

```

1  shared stealableWorkQueue[N]                                /* 每线程一个 */
2  me ← myThreadId
3
4  acquireWork():
5      if not isEmpty(myMarkStack)                            /* 本地标记栈存在待处理工作 */
6          return
7      lock(stealableWorkQueue[me])
8      /* 将本地可窃取工作列表中的一半元素迁移到标记栈 */
9      n ← size(stealableWorkQueue[me]) / 2
10     transfer(stealableWorkQueue[me], n, myMarkStack)
11     unlock(stealableWorkQueue[me])
12
13     if isEmpty(myMarkStack)
14         for each j in Threads
15             if not locked(stealableWorkQueue[j])
16                 if lock(stealableWorkQueue[j])
17                     /* 将该线程可窃取工作列表中的一半元素迁移到标记栈 */
18                     n ← size(stealableWorkQueue[j]) / 2
19                     transfer(stealableWorkQueue[j], n, myMarkStack)
20                     unlock(stealableWorkQueue[j])
21                 return
22
23 performWork():
24     while pop(myMarkStack, ref)
25         for each fld in Pointers(ref)
26             child ← *fld
27             if child ≠ null && not isMarked(child)
28                 setMarked(child)
29                 push(myMarkStack, child)
30
31 generateWork():
32     /* 将本地标记栈中的所有元素迁移到本地可窃取工作队列中 */
33     if isEmpty(stealableWorkQueue[me])
34         n ← size(markStack)
35         lock(stealableWorkQueue[me])
36         transfer(myMarkStack, n, stealableWorkQueue[me])
37         unlock(stealableWorkQueue[me])

```

所有并行回收器都必须仔细考虑如何进行位图标记，以及如何对大数组进行处理。设置位图中某一位的操作必须满足原子化要求。最显而易见的方法可能是对需要设置的位所在的字加锁，但 Endo 等人提出了一种更加高效的算法，即先简单地读取并检测该标记位，如果其尚未设置，则尝试发起原子化的设置操作，如果该操作失败，则进行重试（由于位图中的标记位仅可能在标记阶段被设置，因而重试次数必然存在上限），如算法 14.2 所示。当然，如果将标记位存放在对象头部，则标记位的设置便无需借助于任何原子操作，如 Flood 等 [2001] 的文献所描述的那样。

算法 14.2 基于位图的并行标记

```

1  setMarked(ref):
2      oldByte ← markByte(ref)
3      bitPosition ← markBit(ref)
4      loop
5          if isMarked(oldByte, bitPosition)
6              return

```

```

7      newByte ← mark(oldByte, bitPosition)
8      if CompareAndSet(&markByte(ref), oldByte, newByte)
9          return

```

经验表明, 大型指针数组通常会引发诸多问题。例如, 为避免标记栈溢出, Boehm 和 Weiser [1988] 会将大对象拆分为多个较小的片段 (128 个字) 来进行处理。类似地, Endo 等人在将大对象添加到标记栈或者工作队列之前, 会将其拆分成 512 字节的片段来确保线程之间的负载均衡, 此时标记栈或者工作队列中所记录的将是 <address, size> 对组。

Flood 等人 [2001] 所设计的并行分代回收器通过复制策略来管理年轻代, 同时使用标记-整理策略来管理年老代。本节我们仅关注其并行标记部分。在 Endo 等人的设计中, 每个线程都配备了一个标记栈以及一个可窃取工作队列, 而在 Flood 等人的设计中, 他们使用 Arora 等人 [1998] 提出的算法实现无锁工作窃取, 且每个回收线程仅需使用一个双端队列, 该算法的同步开销较低, 能够支持单个对象级别的负载均衡。该算法的工作流程如下 (也可参见 13.8 节 Arora 并发队列的实现细节, 即算法 13.38): 回收线程将其自身双端队列的底部当作标记栈, 对于该线程而言, push 操作无需任何同步操作, pop 操作只有在队列中只剩一个元素时才需使用同步操作。处于空闲状态的线程将使用同步 remove 操作窃取其他线程双端队列的顶端元素。这一工作窃取策略的优点之一是, 只有当线程之间需要进行负载均衡时才会引入同步操作, 而反观其他工作窃取方案 (例如我们即将介绍的灰色工作包算法), 其负载均衡开销存在于算法的全部执行过程中。

为避免在回收过程中进行动态内存分配, Flood 等人的线程本地双端队列长度固定, 但这存在潜在的队列溢出风险。针对这一问题, 他们提供了额外的全局溢出集合, 该集合仅会在每个类上引入很小的开销。在他们的设计中, 每个 Java 类所对应的类型信息都会持有一个链表, 该链表是由该类型的所有溢出实例链接而成的 (如图 14.2 所示)。将溢出对象彼此链接的指针会复用对象头部指向其类型信息的指针, 尽管该指针会在对象从溢出链表中移除时得到恢复, 但这一修改行为决定了该算法不适用于并发回收器, 因为对于并发回收而言, 回收过程中赋值器可能需要访问对象的类型信息。该算法的溢出处理逻辑如下: 一旦回收线程在将某一元素添加到本地双端队列底部时发生溢出, 则其会将队列中的一半元素迁移到它们各自所属类型的溢出集合中。相应地, 处于空闲状态的线程在发起工作窃取之前, 会先尝试使用溢出集合中的元素来将其本地双端队列填充到半满状态 (如算法 14.3 所示)。

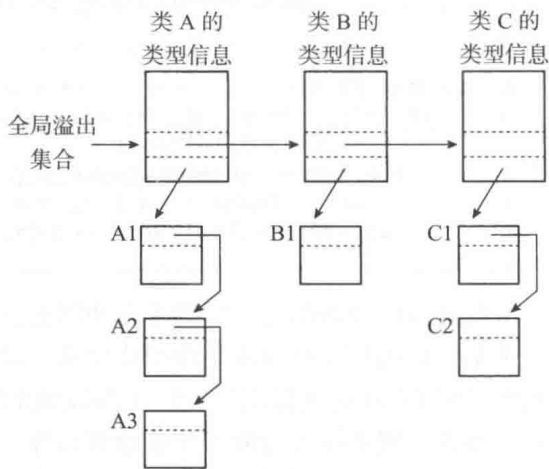


图 14.2 以两级链表方式实现的全局溢出集合 [Flood 等, 2001]。每个 Java 类的类型信息都会持有其所对应溢出实例的链表头, 而每个溢出实例则会复用其头部的类型指针来实现链接

算法 14.3 Flood 等人 [2001] 的并行标记-清扫算法

```

1  shared overflowSet
2  shared deque[N]

```

/* 每线程一个 */

```

3  me ← myThreadId
4
5  acquireWork():
6      if not isEmpty(deque[me])
7          return
8      n ← size(overflowSet) / 2
9      if transfer(overflowSet, n, deque[me])
10         return
11     for each j in Threads
12         ref ← remove(deque[j])           /* 尝试从编号为 j 的队列中窃取 */
13         if ref ≠ null
14             push(deque[me], ref)
15         return
16
17 performWork():
18     loop
19         ref ← pop(deque[me])
20         if ref = null
21             return
22         for each fld in Pointers(ref)
23             child ← *fld
24             if child ≠ null && not isMarked(child)
25                 setMarked(child)
26                 if not push(deque[me], child)
27                     n ← size(deque[me]) / 2
28                     transfer(deque[me], n, overflowSet)
29
30 generateWork():
31     /* 空 */

```

Siebert[2010] 在 Jamaica 实时 Java 虚拟机的并行与并发实现中同样使用了工作窃取技术。为限制标记阶段每个工作步骤的执行时间，Jamaica 将对象拆分为通过指针彼此链接的内存块，且回收器以内存块而非对象为最小工作单元，因此该方案会将每个内存块与一种颜色相关联。我们将在第 15 章看到，并发执行的赋值器线程和回收器线程都可能需要对灰色内存块链表进行访问[⊖]。为避免这一场景下的同步开销，Jamaica 虚拟机使用处理器本地灰色链表。此时对象内部的链表指针将不能复用指向类型信息的指针，因而 Siebert 的策略是复用表示对象颜色的字，并使用 CompareAndSwap 操作将对象链入本地灰色链表中。但这一方案的开销在于，内存块的颜色必须使用一个完整的字而非数个位来表示。为确保负载均衡，处于空闲状态的线程将尝试窃取另一个线程灰色链表中的全部元素。为避免两个线程处理相同的灰色对象，Siebert 使用煤灰色（anthracite）来表示线程正在扫描的对象。线程还可以通过另一种策略实现工作窃取，即尝试将另一个处理器本地灰色链表的表头修改为煤灰色。这一窃取机制粒度较粗，其最佳适用场景应当是被窃取线程只会生成新的工作但不处理任何工作的情形。实时并发回收器中可能会存在此类线程。但如果所有线程都会处理回收工作，则可能出现所有线程全部都竞争仅有的一个灰色内存块链表的情况。Siebert 声称这一情况在实践中通常不会发生。

282

工作窃取场景下的结束检测。回收器必须能够判定某一工作阶段何时结束，即在何时所有协同工作的线程都已执行完毕。Endo 等人 [1997] 最初尝试使用一个全局的计数器来表示空标记栈和空可窃取标记队列的数量，但原子更新该计数器的操作将导致结束检测串行化，

⊖ 在并发回收器中，赋值器线程可能需要承担一定的回收工作。——译者注

[283]

在大型系统（处理器数量达到 32 个甚至更多）中加锁操作可能会产生显著的时间开销。针对这一问题，他们的解决方案是为每个处理器关联两个旗标，即栈空旗标和队列空旗标，分别表示标记栈或者可窃取工作队列是否为空，线程在设置或清除这些旗标时无需使用同步操作，具体实现细节可参见算法 13.18。在执行结束检测时，检测线程先清空全局检测中止（detection-interrupted）旗标，然后顺次检测所有其他处理器的空闲旗标，最后再判断检测中止旗标是否被其他处理器重新设置，如果没有，则意味着该阶段结束。该策略要求处理器 A 在窃取处理器 B 的全部工作时必须严格遵守如下协议：首先清空自身的栈空旗标，然后设置检测中止旗标，最后设置 B 的队列空旗标。但 Petrank 和 Kolodner 指出，该算法不允许多个线程同时进行结束检测，因为第二个检测线程可能会在第一个检测线程设置检测中止旗标之后再次将其清空，从而导致第一个检测线程误认为该旗标始终处于清空状态。

Kolodner 和 Petrank [1999] 提出了一种能够处理多种并发问题的通用解决方案。为确保在任意时刻只有一个线程可以执行结束检测，他们引入了一个全局字来作为检测器身份标识。线程在尝试进行结束检测之前必须先判定该身份标识是否为 -1（意味着当前没有线程执行结束检测），如果结果为真，则尝试原子化地设置身份标识，否则，将继续等待。

Flood 等人使用一个状态字来执行结束检测，该状态字的每个位对应一个回收线程，且对该字进行修改时必须使用原子操作。在初始情况下，所有线程均处于活动状态，一旦某一线程完成工作（且未从其他线程窃取到任何工作），则其会把自身标记位清空，并循环检测状态字中的所有状态标记位是否都被清空。如果结果为真，意味着所有线程均完成工作，即回收阶段结束；否则，线程将尝试从其他线程窃取工作。如果存在可窃取工作，则线程先将自身状态位设置为活跃，然后尝试窃取；如果窃取失败，其将再次清空自身状态位并继续执行循环检测。该算法显而易见的问题在于回收线程的数量不得超过一个字中所包含的位的数量，为此，Flood 等人建议使用活跃线程计数器来替代状态字。

灰色工作包（grey packets）。Ossia 等人发现，具备工作窃取能力的标记栈最适用于回收线程数量可以提前预知的回收器 [Ossia 等，2002；Barabash 等，2005]，但是对于要求赋值器（在分配过程中）也承担一小部分回收工作的场景，该策略可能不再适用。除此之外他们认为，如果线程既要选择最佳目标队列来进行工作窃取，又要执行结束检测，实现起来可能存在困难。针对这一问题，他们的负载均衡策略是驱使每个线程竞争标记工作包以进行处理。在他们的系统中，可用工作包的数量固定（1000 个），且每个工作包的大小也保持固定（512 个元素）。

[284]

每个线程使用两个工作包，线程本身对输入工作包中的元素进行处理，并将新产生的工作添加到输出工作包中。在三色抽象框架下，两种工作包中的元素均为灰色，因而我们使用灰色工作包（grey packets）这一名称，该名称最初是由 Thomas 等人 [1998] 在 Insignia's Jeode Java 虚拟机中提出的^①。线程会尝试从全局工作池中获取新的工作包，当线程填满输出工作包时会将其释放到全局工作池，并从中获取一个新的空工作包。Ossia 等在全局工作池中维护三个工作包链表，它们分别是如下 3 种工作包的集合：空工作包、填充率不足一半的工作包、接近填满的工作包，如图 14.3 所示。线程在获取输入工作包时将优先选择接近填满的工作包（算法 14.4 中的 `getInPacket` 方法），而在获取输出工作包时将优先选择空工作包（算法 14.4 中的 `getOutPacket` 方法）。

① 这一思想最初是由 Ossia 等人 [2002] 提出的，但他们并未对其申请专利。

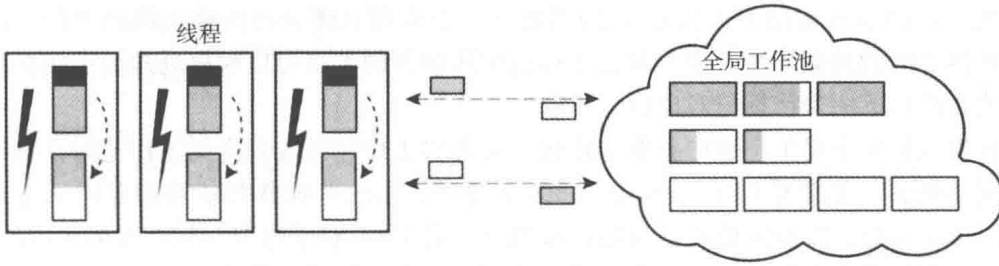


图 14.3 灰色工作包。每个线程会从全局工作池中获取一个空工作包来替代已被填满的输出工作包。线程在标记过程中会使用新的待追踪引用填充输出包，一旦填满，则将其与全局工作池中的空工作包进行置换

算法 14.4 灰色工作包的管理

```

1  shared fullPool                                /* 满工作包全局池 */
2  shared halfFullPool                            /* 半满工作包全局池 */
3  shared emptyPool                              /* 空工作包全局池 */
4
5  getInPacket():
6      atomic
7          inPacket ← remove(fullPool)
8      if isEmpty(inPacket)
9          atomic
10             inPacket ← remove(halfFullPool)
11     if isEmpty(inPacket)
12         inPacket, outPacket ← outPacket, inPacket
13     return not isEmpty(inPacket)
14
15 testAndMarkSafe(packet):
16     for each ref in packet
17         safe(ref) ← allocBit(ref) = true        /* 私有数据结构 */
18
19
20 getOutPacket():
21     if isFull(outPacket)
22         generateWork()
23     if outPacket = null
24         atomic
25             outPacket ← remove(emptyPool)
26     if outPacket = null
27         atomic
28             remove(halfFullPool)
29     if outPacket = null
30         if not isFull(inPacket)
31             inPacket, outPacket ← outPacket, inPacket
32     return
33
34 addOutPacket(ref):
35     getOutPacket()
36     if outPacket = null || isFull(outPacket)
37         dirtyCard(ref)
38     else
39         add(outPacket, ref)

```

灰色工作包策略存在诸多优势。在将标记过程的输入和输出相分离之后（Ossia 等避免

直接交换线程的输入输出工作包), 处理器通常不会立即处理自己所输出的新工作, 因而实现了处理器之间的负载均衡。由于灰色工作包中所包含的都是可以顺序处理的引用队列, 所以其天然支持对下一个待标记对象进行预取。

只有当线程从全局工作池中获取工作包, 或者将工作包释放到全局工作池时, 算法才需要引入同步操作。如果借助于 CompareAndSwap 原语, 则这些操作都能够以非阻塞方式实现(为避免 ABA 问题, 线程需要将自身标识添加到链表头)。对于内存一致性保障较弱的系统, 它们还通过多种策略来尽量减少内存屏障的使用。只有在线程获取或者释放工作包时才需引入内存屏障, 而标记过程以及将单个对象压栈的过程都无需引入屏障。在对线程栈进行保守式扫描的过程中, Ossia 等人使用分配位(allocation bits)向量来判定潜在的“指针”是否真正指向某个已分配对象, 分配位向量也可用于赋值器与回收器之间的同步。在 Ossia 等人的设计中, 分配器使用本地分配缓冲区, 一旦本地分配缓冲区溢出, 则分配器首先执行内存屏障, 然后再为本地分配缓冲区中的所有对象设置分配位, 最后才会申请新的本地分配缓冲区。这一顺序可以确保在从新缓冲区中分配对象之前, 老缓冲区中的对象都已正确设置分配位(如算法 14.5 所示)。算法还有两处需要引入屏障, 一处是追踪线程在获取新的输入工作包时, 此时线程需要检查新工作包中每个对象的分配位, 并在某一私有数据结构中记录每个对象是否可以安全进行扫描(即其分配位是否已被设置)。该过程完成后, 线程首先执行内存屏障, 然后进一步对所有安全对象进行追踪。不安全对象会被添加到额外的工作包中, 回收器将延迟对这些对象的追踪(见算法 14.6), 延迟工作包有可能会在某一时刻释放到全局工作池中。这一策略可以确保线程不会追踪分配位尚未设置的对象。追踪线程在将其输出工作包释放到全局工作池时也需执行内存屏障(其目的在于避免处理器指令重排序可能产生的错误, 即线程在将工作包释放到全局工作池之后依然向其中追加工作)。从全局工作池中获取输入工作包的操作通常无需引入屏障, 因为加载工作包指针和对该指针进行访问这两个操作之间存在依赖加载关系, 绝大多数硬件都可以确保该操作的执行顺序。

算法 14.5 灰色工作包策略中的并行分配

```

1 sequentialAllocate(n):
2     result ← free
3     newFree ← result + n
4     if newFree ≤ labLimit
5         free ← newFree
6     return result
7
8     /* 本地分配缓冲区溢出 */
9     fence
10    for each obj in lab
11        allocBit(obj) ← true
12    lab, labLimit ← newLab()
13    if lab = null
14        return null
15    sequentialAllocate(n)

```

\$

/* 内存耗尽 */

算法 14.6 基于灰色工作包的并行扫描

```

1 shared fullPool
2
3 acquireWork():
4     if isEmpty(inPacket)

```

/* 满工作包全局池 */

```

5      if getInPacket()
6          testAndMarkSafe(inPacket)
7          fence
8
9  performWork():
10     for each ref in inPacket
11         if safe(ref)
12             for each fld in Pointers(ref)
13                 child ← *fld
14                 if child ≠ null && not isMarked(child)
15                     setMarked(child)
16                     addOutPacket(child)
17             else
18                 addDeferredPacket(ref)          /* 延迟对不安全对象的追踪 */
19
20 generateWork():
21     fence
22     add(fullPool, outPacket)
23     outPacket ← null

```

灰色工作包的状态跟踪相对容易。每个全局工作池都会关联一个表示其内部工作包数量的计数器，线程在获取或者释放工作包之后会通过原子操作对其进行更新。由于计数器的更新会比工作包的获取或者释放操作滞后，所以该值只能作为工作包数量的一个近似。但无论如何，一旦空工作包的数量与工作包的总量相同，则必然意味着扫描阶段的结束。为确保空工作包的数量与工作包总量不会临时性地相等，每个线程在置换工作包时必须遵循先获取新工作包、再释放老工作包的顺序。在回收开始时，线程必须先获取其输入工作包、再获取输出工作包，这样才能确保即使线程没有发现任何可以处理的工作，结束检测也不会受到影响。如果各回收线程均遵从上述两个规则，则获取 / 释放工作包与更新工作包计数这两个操作便可异步地执行。

灰色工作包策略限制了标记队列的整体最大深度，所以标记过程可能会溢出。如果线程无法获取尚未填满的输出工作包，则其会交换自身输入工作包和输出工作包的角色，但如果这两个工作包也被填满，则需要借助于额外的溢出处理机制。Ossia 等人的解决方案是继续进行对象标记，但不将新发现的子节点添加到任何输出工作包中，与此同时，他们会将这些对象所对应的卡表打上脏标记。回收器稍后会对卡表进行扫描，进而找出所有包含未标记子节点的已标记对象。当然，也可像 Flood 等人 [2001] 那样将溢出对象链接到其类型所对应的类结构上。

多通道。Wu 和 Li[2007] 提出了一种针对大规模处理器的负载均衡策略，该策略无需依赖昂贵的原子操作。在他们的方案中，线程之间通过单生产者、单消费者通道（参见算法 13.34）来交换标记任务，如算法 14.7 所示。对于包含 P 个标记线程的系统，每个线程都会配备 $P-1$ 个队列，其实现方式为环状缓冲区。通道中值为 `null` 的槽意味着空槽。基于这一规则，单生产者、单消费者通道无需任何昂贵的原子操作。对于处理器数量巨大的服务器，该算法的性能要优于 Flood 等人 [2001] 的工作窃取算法。

算法 14.7 基于通道的并行追踪

```

1  shared channel[N,N]          /* N × N 个单生产者、单消费者通道 */
2  me ← myThreadId
3
4  acquireWork():

```

285
?
287

288

```

5     for each k in Threads
6         if not isEmpty(channel[k,me])           /* 线程 k 给我推送了工作 */
7             ref ← remove(channel[k,me])
8             push(myMarkStack, ref)               /* 将其压入自身标记栈中 */
9             return
10
11 performWork():
12     loop
13         if isEmpty(myMarkStack)
14             return
15         ref ← pop(myMarkStack)
16         for each fld in Pointers(ref)
17             child ← *fld
18             if child ≠ null && not isMarked(child)
19                 if not generateWork(child) /* 推送工作到其他线程 */
20                     push(myMarkStack, child)
21
22 generateWork(ref):
23     for each j in Threads
24         if needsWork(k) && not isFull(channel[me,j])
25             add(channel[me,j])
26             return true
27     return false

```

与 Endo 等 [1997] 所使用的策略类似, 该算法会将工作任务主动推送给其他线程。当某一线程 i 生成新的工作任务之后, 它首先会检查是否存在某一线程 j 需要新的工作, 如果结果为真, 则将该任务添加到输出通道 $\langle i \rightarrow j \rangle$, 否则, 其会将该任务压入自身标记栈中。一旦线程 i 的标记栈为空, 则其会尝试从某一输入通道 $\langle j \rightarrow i \rangle$ 获取工作。不幸的是, 自身不产生任何新标记工作的线程无法将工作分摊给其他空闲线程, 针对这一情况, 线程会将自身栈底的工作任务推送给等待工作的线程。Wu 和 Li 声称, 这一负载均衡策略可以将所有线程保持在繁忙状态。通道长度的选择取决于线程使用自身标记栈的繁忙程度, 以及线程是否需要寻求新的工作。如果线程不会频繁寻求工作, 则应选择较小的通道长度。对于包含 16 个 Intel Xeon 处理器的服务器, 通道长度为 1 或者 2 时表现最佳。他们所使用的结束检测机制类似于 Kolodner 和 Petrunk [1999] 采用的, 但为避免多线程同时进行结束检测时可能存在的问题, 他们指定某一线程作为检测线程。

14.6 并行复制

并行标记算法所遇到的大多数问题都会在并行复制算法中出现, 但正如我们曾经提到过的, 两者之间的最大区别在于, 多次对同一对象进行标记通常不会出现问题, 而为同一对象创建多个副本则通常无法接受。我们将依次介绍以处理器为中心和以内存为中心的并行复制策略。

14.6.1 以处理器为中心的并行复制

处理器之间的工作划分。Blelloch 和 Cheng 在副本复制回收 (replicating collection) [Blelloch and Cheng, 1999; Cheng and Blelloch, 2001; Cheng, 2001] 中实现了并行复制。副本复制回收器的具体细节我们将在第 17 章讨论, 简而言之, 该回收器属于增量回收器或者并发回收器, 该回收器会在赋值器执行的同时复制存活对象, 且回收器特别注意将可能会在回收过程中被赋值器并发更新的域修正到正确的值。本章我们仅讨论其设计中的并行部分。

每个复制线程都拥有其本地工作栈, Blelloch 和 Cheng 声称, 与基于 Cheney 队列的复

制策略相比，基于栈的算法能够简化复制线程之间的同步，且其碎片化程度更低（稍后我们将介绍基于 Cheney 队列的并行复制回收器）。在回收过程中，线程会周期性地在本地栈和全局栈中交换工作，并据此实现负载均衡（如算法 14.8 所示）。我们曾经提到，对于简单的共享栈，其压入和弹出操作必须使用同步操作，同时也不存在一种类似于 FetchAndAdd 原语的操作可以原子化地在增加 / 减少栈指针的同时插入 / 弹出元素。我们当然可以使用锁或者基于 LoadLinked/StoreConditionally 操作实现顺序访问，但除此之外，我们也可基于这些指令实现在同一时间内，所有线程都只向栈中压入元素，或者都只从栈中弹出元素，即：要么所有线程都只增加栈指针，要么都只减少栈指针。一旦线程成功移动了栈指针（可能涉及若干个槽），则该线程对这些栈槽的访问将不存在任何竞争风险。

算法 14.8 Blleloch 和 Cheng[2001] 的并行复制算法

```

1  shared sharedStack                                /* 共享工作栈 */
2  myCopyStack[k]                                    /* 本地工作栈最多包含 k 个槽 */
3  sp ← 0                                             /* 本地栈指针 */
4
5  while not terminated()
6      enterRoom()                                    /* 进入弹出工作空间 */
7      for i ← 1 to k
8          if isLocalStackEmpty()
9              acquireWork()
10             if isLocalStackEmpty()
11                 break
12             performWork()
13             transitionRooms()
14             generateWork()
15             if exitRoom()                            /* 离开弹出工作空间 */
16                 terminate()
17
18  acquireWork():
19      sharedPop()                                    /* 从共享栈中迁移工作到本地栈 */
20
21  performWork():
22      ref ← localPop()
23      scan(ref)                                       /* 参见算法 4.2 */
24
25  generateWork():
26      sharedPush()                                    /* 将本地栈中的工作迁移到共享栈 */
27
28  isLocalStackEmpty()
29      return sp = 0
30
31  localPush(ref):
32      myCopyStack[sp++] ← ref
33
34  localPop():
35      return myCopyStack[--sp]
36
37  sharedPop():                                       /* 从共享栈中迁移工作到本地栈 */
38      cursor ← FetchAndAdd(&sharedStack, 1)         /* 尝试从共享栈中获取元素 */
39      if cursor ≥ stackLimit                          /* 共享栈为空 */
40          FetchAndAdd(&sharedStack, -1)             /* 调整栈 */
41      else
42          myCopyStack[sp++] ← cursor[0]              /* 将工作迁移到本地栈 */
43

```

```

44 sharedPush():                               /* 将本地栈中的工作迁移到共享栈 */
45     cursor ← FetchAndAdd(&sharedStack, -sp) - sp
46     for i ← 0 to sp-1
47         cursor[i] ← myCopyStack[i]
48     sp ← 0

```

289
290

Blelloch 和 Cheng[1999] 引入了“工作空间”(room)这一概念来控制线程对共享栈的访问, 他们规定: 在任意时刻, 弹出工作空间(push room)和压入工作空间(pop room)中至少有一个必须为空, 其整体工作流程如算法 14.9 所示。在回收内部循环的每次迭代过程中, 线程首先进入弹出工作空间, 然后处理固定的工作量, 这些工作既可能来自于线程本地栈, 也可能是线程使用 FetchAndAdd 操作从共享栈中获取的。线程会将自身新生成的工作压入本地栈。处理结束后, 线程将离开弹出工作空间并尝试进入压入工作空间, 但在此之前, 其必须等待所有其他线程都离开弹出工作空间。第一个进入压入工作空间的线程将关闭 gate 变量以避免其他线程进入弹出工作空间。在进入压入工作空间之后, 线程将把本地工作栈中的所有元素释放到共享栈, 该过程仍需使用 FetchAndAdd 操作在共享栈上预留空间。最后一个离开压入工作空间的线程将打开 gate 变量。

该策略的缺陷在于, 所有尝试进入压入工作空间的线程都必须等待所有位于弹出工作空间的线程完成其正在处理的任务。与获取或者释放新工作相比, 将对象着为灰色的工作量相当可观, 且线程在进入压入阶段之前, 必须等待所有其他线程都完成弹出阶段的着色工作。如果各线程处理着色工作的时间差异较大, 则会造成很大的处理器资源浪费。更加富有弹性的抽象可能是允许线程在离开弹出工作空间之后不进入压入工作空间, 此时线程将处理自身新生成的、位于本地工作栈中的灰色对象, 这一过程无需与共享栈发生任何交互, 也无需进入任意一个工作空间。这一策略极大地提高了弹出工作空间为空的概率, 从而减少了线程在进入压入工作空间之前的期望等待时间。

291

Blelloch 和 Cheng 的原始工作空间抽象算法支持简单直接的结束检测: 当线程离开压入工作空间之后, 其自身工作栈必然为空, 因而一旦最后一个离开压入工作空间的线程发现共享栈为空, 则意味着该阶段结束。但在更加松散的抽象策略中, 回收线程可能会在工作空间之外处理回收工作, 因而回收器必须为共享栈维护一个全局计数器, 该计数器的值表示有多少线程正在处理从共享栈中获取的工作。最后一个离开弹出工作空间的线程必须检测该计数器是否为零以及共享栈是否为空, 如果结果为真, 则意味着该阶段结束。

算法 14.9 基于工作空间的压入 / 弹出同步

```

1  shared gate ← OPEN
2  shared popClients                               /* 弹出工作空间中的线程数量 */
3  shared pushClients                             /* 压入工作空间中的线程数量 */
4
5  enterRoom():
6      while gate ≠ OPEN
7          /* 等待 */
8      FetchAndAdd(&popClients, 1)                 /* 试着开始从共享栈中弹出工作 */
9      while gate ≠ OPEN
10         FetchAndAdd(&popClients, -1)             /* 未进入成功, 撤销变更 */
11         while gate ≠ OPEN
12             /* 等待 */
13         FetchAndAdd(&popClients, 1)               /* 重试 */
14
15

```



```

16 transitionRooms():
17     gate ← CLOSED
18     FetchAndAdd(&pushClients, 1)                /* 从弹出状态切换到压入状态 */
19     FetchAndAdd(&popClients, -1)
20     while popClients > 0
21         /* 等待其他线程离开弹出工作空间 */
22
23 exitRoom():
24     pushers ← FetchAndAdd(&pushClients, -1) - 1    /* 停止压入 */
25     if pushers = 0    /* 本线程最后一个离开压入工作空间, 执行结束检测 */
26         if isEmpty(sharedStack)    /* 所有灰色对象都已处理完毕 */
27             gate ← OPEN
28             return true
29         else
30             gate ← OPEN
31             return false

```

对象复制的并行化。为确保单个对象仅会被一个线程复制，线程之间必须争夺待复制对象并在其旧版本的头部填充转发地址。线程复制对象的方式取决于它们之间是否共享同一个分配区域。共享分配区域可以减少内存浪费，但线程在执行分配时必须使用原子操作。在使用共享分配区域的策略下，Blelloch 和 Cheng[1999] 要求线程以竞争的方式向对象转发指针域中写入某个意味着“待复制”的值，胜出的线程将有权进行对象的复制并将新副本的地址写入老对象的转发指针域，而竞争失败的线程则必须等待转发地址槽中的值成为一个有效指针。另外，如果线程可以预知新对象的地址（例如将对象复制到本地分配缓冲区），则线程可以在执行复制之前以竞争的方式写入转发地址。

Marlow 等 [2008] 在 GHC Haskell 系统中比较了两种复制策略。其第一种复制策略是使用共享分配区域：线程在复制对象之前首先判断该对象是否已被转发，如果结果为真，则返回其转发地址，否则，其将尝试使用 CompareAndSwap 操作将 busy 值写入该对象的转发指针域，且 busy 值应当能够与该域中可能存在的正常值（例如锁或者哈希值）或者有效转发地址相区分；如果该操作成功，则线程将复制该对象、写入转发地址并返回目标空间中新副本的地址；如果写入 busy 值的 CompareAndSwap 操作失败，则线程将陷入自旋，直到竞争胜出的线程完成该对象的复制。第二种策略是使用线程本地分配缓冲区：线程先乐观地将对象复制到本地分配缓冲区，然后再使用 CompareAndSwap 操作更新转发地址，如果 CompareAndSwap 操作失败，则复制操作必须撤销（例如，将线程本地分配缓冲区的空闲指针恢复为原有值）。Marlow 等发现后一种策略的性能稍高，其原因在于竞争极少出现。同时他们建议，在第二种场景下，如果对象不可修改且可以容忍少许重复，则在写入转发地址时可以使用非同步操作来替代原子操作。

本章曾经提到，在 Flood 等人 [2001] 所实现的分代回收器中，其年老代使用标记-整理算法管理，年轻代使用复制式算法管理，且两种算法都实现了并行化。我们已经介绍过其并行标记的实现机制，此处我们将介绍其并行复制算法。用于记录待扫描对象的可窃取工作队列将在此处得到复用，但与并行标记相比，并行复制需要额外关注两个问题：一是如何将并行复制过程中由于内存分配所产生的竞争最小化，二是如何确保每个存活对象仅被复制一次。对于前一个问题，Flood 等的解决方案是使用本地分配缓冲区（参见 7.7 节），年轻代内部的存活对象复制以及从年轻代到年老代的提升均采用这一策略；而对于后一个问题，线程在复制过程中首先投机性地将存活对象复制到本地分配缓冲区，然后尝试使用 CompareAndSwap 操作更新其转发地址，如果该操作成功，则意味着复制成功，否则，其将返

回已由其他线程设置的转发地址。

本书多次提到,程序的局部性对其性能会产生很大影响。对于使用非一致内存架构的处理器而言,局部性作用将更加明显,因而在这一情况下,理想的对象迁移策略应当是将对象靠近最频繁访问它们的处理器布置。现代操作系统大都支持标准的内存亲和策略 (memory affinity policie), 该策略将决定把哪些内存地址预留给哪个处理器。典型的内存关联策略有两种:一种是首次访问 (first-touch) 策略,也称本地 (local) 策略,此时线程所申请的内存将在其所运行的处理器所关联的内存中分配;另一种是轮循 (round-robin) 策略,即线程所需的内存将在所有处理器所关联的内存中分配。支持处理器亲和性 (processor-affinity) 的线程调度器会尝试将线程调度到其上一次所运行的处理器上执行。Ogasawara [2009] 发现,即使对于使用本地处理器亲和策略的系统,如果内存管理器无法意识到其运行在非一致内存架构之上,则其依然可能无法将对象分配到合理的位置。如果本地分配缓冲区小于一页,且其以线性方式分发给各线程,则某些线程可能需要在远端内存中执行分配,特别是在操作系统使用大页表 (16MB) 来减少物理地址映射开销的场景下。另外,回收器在移动对象时通常不会考虑其内存亲和性。

相比之下, Ogasawara 所设计的内存管理器可以感知到其正运行在非一致内存系统中,此时内存管理器会以一页或者多页为单位将堆划分为段 (segment), 且每一段都会映射到一个处理器中。在赋值器或者回收器申请内存时,分配器会优先从首选处理器 (preferred processor) 对应的段中分配:对于赋值器而言,首选处理器应当是该线程当前所运行的处理器,而对于回收器而言,如果对象原本就与某一处理器相关联,则回收线程将对对象优先迁移到该处理器所对应的段中。我们并不能保证分配对象的线程将是访问该对象最频繁的线程,因而回收器将使用支配线程信息 (dominant-thread information) 来确定每个对象的首选处理器。首先,对于直接被赋值器线程栈所引用的对象,其首选处理器应当是执行该线程的处理器,且赋值器线程应当周期性地更新自身所对应的首选处理器。其次,回收器可以使用对象的锁相关信息来判定其所对应的支配线程。编程语言中的锁相关语义通常会在对象头部的某个字中保留持有锁的线程身份标识,尽管该信息仅能反映出最后一个对该对象加锁的线程,但是由于许多对象都不会逃逸出其所诞生的线程 (即使对于被上锁的对象也不例外), 因而该方案足以作为获取对象首选处理器的一个近似。最后,回收器可以将父对象的首选处理器传递给其子节点。在图 14.4 所示的案例中,回收器使用 3 个标记线程,为简化问题,我们假定它们均运行在其首选处理器之上,且每个处理器以不同的颜色区分。线程 T0 持有对象 X 的锁,即其已经将自身线程编号写入了对象 X 的头部。我们将图中的每个对象与其首选处理器着为相同的颜色,回收器会将每个对象复制到你其首选处理器的本地分配缓冲区中。

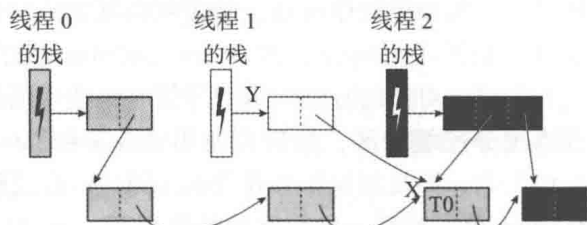


图 14.4 支配线程追踪。为区分起见,我们将线程 0、线程 1、线程 2 分别表示为灰色、白色、黑色,它们已经完成了对象图的追踪。每个对象的颜色预示了其将被复制到哪个处理器的本地分配缓冲区。每个对象的第一个域为其头部。线程 T0 当前持有对象 X 的锁

14.6.2 以内存为中心的并行复制技术

每线程来源空间与目标空间。复制式回收算法天然支持依照对象位置来划分工作,由此

我们可以设计出一种简单的并行复制策略，即为每个回收线程设置本地来源空间与目标空间，并使用 Cheney 式复制策略 [Halstead, 1984]。在该策略中，每个线程都会负责一段连续内存空间的扫描，但其复制对象与设置转发指针的操作仍会与其他线程产生竞争。这一简单策略存在两个明显缺陷：第一，在某一处理器完成所有任务之后，其他处理器仍可能处于工作状态，从而不利于处理器之间的负载均衡；第二，有可能存在一个线程的目标空间溢出、但其他线程仍存在可用目标空间的情况。

块结构堆。另一种解决方案是对目标空间进行更细粒度的划分，线程则通过竞争的方式获取待扫描内存块以及容纳存活对象的内存块。Imai 和 Tick[1993] 将堆划分为较小的、固定大小的内存片，并且每个线程会获取专属的待扫描内存片以及目标缓冲区内内存片，其复制过程基于 Cheney 指针而非显式工作列表。当目标内存片被填满时，线程会将其释放到全局工作池中，以便其他线程继续对其进行扫描，同时其会从空闲内存管理器申请一个新的内存片来作为复制目标缓冲区。该策略使用两种机制来确保负载均衡。第一，用作复制目标缓冲区的内存片（他们将其称作“堆扩展单位”）通常较小（仅 256 个字）。如果在内存片空间较小的情况下使用线性分配，内存碎片问题可能会十分严重，因为平均每个内存片末端会浪费掉半个对象的空间。为解决这一问题，Imai 和 Tick 使用页簇分配（参见第 7 章）策略来分配小对象，因此每个线程将同时拥有多个目标缓冲区内内存片。大对象则会通过加锁的方式复制到共享缓冲区。

第二，线程之间进行负载均衡的粒度会比内存片更小。每个内存片将被进一步划分为更小的内存块（他们称之为“负载分布单元”），其大小通常为 32 个字。内存块越小，加速效果越明显。在算法执行过程中，每个线程在扫描新的内存块之前，其会放弃某些尚未扫描的内存块并将其释放到全局工作池中，其过程如下：在扫描完一个槽并增加扫描指针后，线程会检查后者是否到达当前扫描内存块的边界；如果结果为真，且下一个待扫描对象小于一个内存块的大小，则线程会将扫描指针移动到当前目标内存块的起始地址。这一策略不但有助于减少线程之间从全局工作池中获取内存块时所产生的竞争，而且可以避免将所有灰色对象都集中在目标内存块中。如果线程刚刚扫描过的内存块与目标内存块之间存在未扫描内存块，线程会将其释放到全局工作池中，以便其他线程处理。图 14.5 展示了算法在这两种情况下的执行过程：在图 14.5a 中，线程的待扫描内存块与目标内存块位于同一个内存片中；而在图 14.5b 中，它们位于不同的内存片中。不论在哪种情况下，除了最后一个未扫描内存块（即目标内存块）之外，其他尚未扫描的内存块都将被释放到全局工作池中。

294

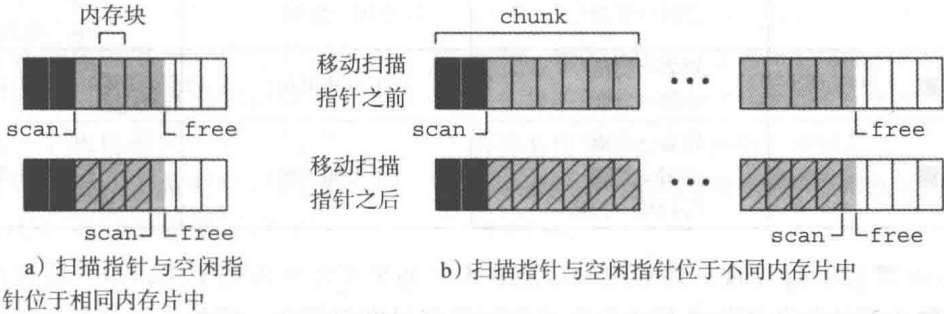




图 14.5 Imai 和 Tick[1993] 并行复制回收器中的内存片管理，图中展示了当线程完成一个内存块的扫描之后扫描指针的变化。带斜线的内存块将被释放到全局工作池中

在线程扫描完一个内存块之后，如果下一个对象大于一个内存块但小于一个内存片，则

线程将其扫描指针移动到目标内存片的起始地址；而对于更大的对象，线程将继续对其扫描，并在复制完成后立即将其释放到全局工作池中。

图 14.6 展示了内存块的状态及其变迁过程[⊖]。处于空闲 (freelist)、待扫描 (scanlist)、完成 (done) 状态的内存块均位于全局工作池中，其他状态的内存块则正由线程进行本地处理。状态变迁箭头上标明了内存块在发生状态变迁时的颜色。在 Imai 和 Tick 的策略中，只有如下 3 个时刻才有可能发生状态转换：scan 指针触达扫描内存块的末端时、copy 指针触达目标内存块的末端时、scan 指针与 free 指针重合时（此时扫描内存块与目标内存块相同，即“别名”）。例如，目标内存块必须存在一定的空闲内存，以便将可达对象复制到其中，因而所有进入复制状态[⊖]的内存块必然未被填满。表 14.1 展示了扫描内存块与目标内存块在不同情况下的状态变迁。例如，如果目标内存块同时包含灰色对象与空闲内存（）且非别名的扫描内存块已经完全成为黑色（），回收线程将把扫描内存块的状态变更为完成，同时继续对目标内存块进行扫描，此时扫描内存块与目标内存块将成为同一个内存块，即两者为别名关系。

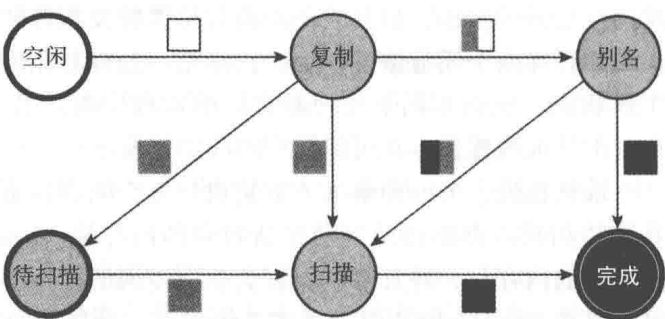


图 14.6 Imai 和 Tick[1993] 回收器中内存块的状态及其变迁过程。如果内存块位于全局工作池中，其必然处于边框较粗的三种状态之一，而正在被线程处理的内存块必然处于边框较细的三种状态之一

表 14.1 Imai 和 Tick 回收器中内存块的状态变迁逻辑

复制内存块 (copy block)	扫描内存块 (scan block)		
	别名	 或 	
	(继续扫描)	(继续扫描)	扫描→完成 复制→别名
	别名→复制 待扫描→扫描	(继续扫描)	扫描→完成 待扫描→扫描
	别名→复制 待扫描→扫描	(不可能出现)	(不可能出现)
	别名→扫描 空闲→复制	复制→待扫描 空闲→复制	扫描→完成 复制→扫描 空闲→复制
	别名→扫描 空闲→复制	(不可能出现)	(不可能出现)
	别名→完成 空闲→复制 待扫描→扫描	(不可能出现)	(不可能出现)

Marlow 等 [2008] 发现，在 GHC Haskell 中，如果所需处理的工作较少，这种以一次扫描一个完整内存块的负载均衡技术可能导致回收器过度串行化。例如，一旦某一线程将其根集合全部复制到单个内存块中，则只有当其扫描指针与空闲指针之间的距离超过一个内存块

⊖ Siegwart 和 Hirzel [2006] 最先提出了这种清晰的记法。
⊖ 即作为对象复制的目标空间。——译者注

[296]

块的状态及其变迁。与图 14.6 类似，处于空闲、待扫描、完成状态的内存块均位于全局工作池中，其他状态的内存块则正由线程进行本地处理。状态变迁箭头上标明了内存块在发生状态变迁时的颜色。表 14.2 展示了扫描内存块与目标内存块在不同情况下的状态变迁。例如，当目标内存块既包含灰色槽又包含空闲内存 (■ 或 □) 、且非别名扫描内存块包含灰色槽时，线程将把扫描内存块释放到全局工作池中，同时继续对目标内存块进行扫描，此时目标内存块与扫描内存块将成为别名关系。因此，Siegwart 和 Hirzel 的状态转换体系可以认为是 Imai 和 Tick[1993] 的超集。

表 14.2 Siegwart 和 Hirzel 回收器中内存块的状态变迁逻辑

复制内存块 (copy block)	扫描内存块 (scan block)		
	别名	■ 或 ■	■
■ 或 ■	(继续扫描)	扫描→待扫描 复制→别名	扫描→完成 复制→别名
□ 或 ■	别名→复制 待扫描→扫描	(继续扫描)	扫描→完成 待扫描→扫描
■ 或 ■	别名→扫描 空闲→复制	复制→待扫描 空闲→复制	扫描→完成 复制→扫描 空闲→复制
■	别名→完成 空闲→复制 待扫描→扫描	(不可能出现)	(不可能出现)

Siegwart and Hirzel [2006], doi: 10.1145/1133956.1133964.

© 2006 Association for Computing Machinery, Inc., 经许可后转载

[297]

在并行复制算法中，回收线程从全局工作池中获取扫描内存块的操作可能会存在较为严重的竞争，为解决这一问题，Siegwart 和 Hirzel 为每个线程额外缓存一个本地扫描内存块。因此“待扫描→扫描”的状态变迁所涉及的内存块将是被缓存的内存块（如果存在的话）；相应地，“扫描→待扫描”的状态变迁会将扫描内存块缓存，新缓存的内存块可能会将老的缓存驱逐到全局待扫描池中。除此之外，他们所使用的内存块（128KB）比 Imai 和 Tick 的更大。并行层次分解复制算法在提升关联对象的空间局部性方面效果显著，大多数父子节点将位于同一个 4KB 的页内，同时该算法还可以减少转译后备缓冲区的访问次数、降低高速缓存不命中的概率。该算法以增大回收器处理时间为代价来换取赋值器的性能提升，这一付出是否真正有效，取决于应用程序、具体实现以及平台。

通道。Oancea 等 [2009] 采用与 Wu 和 Li[2007] 类似的多通道策略（参见第 14.5 节）来避免回收过程对原子同步操作的依赖，但他们的算法却是以内存为中心而非以处理器为中心的。该算法原本的设计目的是提升非一致内存架构下的处理性能，但其在典型的多核平台上同样表现良好。算法将整个堆内存划分为多个分区，且分区的数量远大于处理器的数量。每个分区都存在独立的工作列表，其中所包含的元素是位于该分区目标空间的待扫描对象。每个工作列表在任意时刻都只能由一个处理器进行处理。作者声称，尽管这种将工作列表与内存空间绑定的策略会增加处理器之间的交互开销，但它却可以提升回收过程的硬件友好性。

在回收过程中，处理器依然需要对工作列表中的对象进行扫描。如果新发现的对象位于当前分区，线程便将其添加到自身工作列表中，如果对象位于其他分区，则线程将其引用发送到该分区所对应的工作列表。处理器之间通过单读单写通道进行工作交换，这些通道

的实现方式依然是固定大小的环状缓冲区（参见算法 13.34）。基于 Intel 或 AMD 的 x86 架构，在通道中添加或者移除元素无需借助于任何形式的锁或者其他昂贵的内存屏障，但对于 PowerPC 等不提供强访问顺序保障的架构，要么必须引入屏障，要么必须通过某种协议确保缓冲区中每个空槽的值均为 `null`。线程只有在尝试获取一个分区 / 工作列表时才需使用锁。分区的大小为 32KB，这一空间单元比之前所介绍的其他算法都要大。与更细粒度的划分策略相比，较大的分区粒度降低了处理器之间的交互开销，但其却不利于处理器之间的负载均衡。

在回收过程中，每个线程会对其输入通道和工作列表中的元素进行处理，每个线程的结束条件是：①其不再拥有任何工作列表；②其输入输出通道皆为空；③所有线程的所有工作列表均为空。每个线程在自身回收结束时都会设置全局可见旗标。Oancea 等通过一种具有实用性的策略来管理回收器，即在回收开始阶段，他们先使用经典的追踪算法完成 30 000 个对象的处理，然后再将此时的灰色对象加入到各自所在区域对应的工作列表中，接下来再将处理算法切换到基于通道的模式，并将各工作列表分摊给各处理器进行处理。

卡表 (card table)。分代回收器通常会使用并行复制技术来管理其年轻代，但这一策略又会引入另一个问题，即如何对记忆集中的根进行并行处理。记忆集的实现方式可以是缓冲区链表、哈希表、卡表，前两种实现方式均可以使用我们曾经所描述过的技术进行处理。例如，如果记忆集的实现方式是缓冲区链表，则线程之间可以采用与块结构算法类似的方式来竞争下一个缓冲区，并据此实现负载均衡。但对于以卡表方式实现的记忆集，线程之间的负载均衡则要复杂的多。在对年轻代进行回收时，回收器必须对卡表所指示的区域进行扫描，进而才能找到所有潜在的分代间指针。一种显而易见的并行扫描策略是将卡表划分为连续的、大小相等的内存块，并且静态地将每个内存块与一个回收线程相关联，或者各线程以动态竞争的方式获取卡。但是，存活对象在各内存块中的分布通常并不均匀，即某些内存块中的存活对象可能十分密集，而在另一些内存块中则十分稀疏。Flood 等 [2001] 发现这一简单的工作划分方法通常无法有效实现负载均衡，因为对存活对象较为密集的内存块进行扫描通常会占据大多数回收时间。为解决这一问题，他们将卡表进一步细分为 N 个步 (stride)，同一步中的每个卡之间相距 N 个卡，即：卡集合 $\{0, N, 2N, \dots\}$ 组成一步，卡集合 $\{1, N+1, 2N+1, \dots\}$ 组成下一步，以此类推。这一策略会将存活对象较为平均地散布于各个步中，同时各线程在对卡表进行扫描时会以竞争的方式获取步而非内存块。

298

14.7 并行清扫

本章的最后我们将介绍并行清扫与并行整理算法。这两种算法均是对追踪完成之后的对象进行处理，此时回收器已完成堆中所有存活对象的识别，因而这两个阶段的并行化都相对简单。

从原则上讲，清扫阶段的并行化解决方案十分简单，要么将堆静态地划分为多个连续内存块，要么将堆划分为更细粒度的内存块，各线程通过竞争的方式来获取待清扫内存块，并将完成处理的内存块插入空闲链表。但在该策略中，将空闲内存插入空闲链表的操作只能串行，从而可能成为清扫过程的瓶颈。幸运的是，几乎在所有并行系统中，处理器都可以拥有本地空闲链表并使用分区适应分配（参见第 7 章），因此各线程之间的竞争将仅发生在将空闲内存块归还给全局内存块分配器的过程中。另外，懒惰清扫（参见第 2 章）天然就是一种并行清扫解决方案，它可以根据赋值器线程的分配率天然实现对半满内存块清扫的负载均衡。

懒惰清扫所执行的第一个步骤（也是唯一一步）是识别出完全为空的内存块并将其返回给内存块分配器。为减少这一过程中的竞争，Endo 等 [1997] 为每个回收线程维护数个（例如 64 个）连续内存块以进行本地处理。其回收器使用位图标记，位图本身位于内存块的头部，而其头部会相对内存块本身独立存放。基于这一策略，回收线程可以十分简单地判定某一内存块是否完全为空，然后再将完全为空的内存块排序、合并，最后将其添加到本地空闲内存块链表。不完全为空的内存块则将添加到本地回收链表，以便赋值器线程进行懒惰清扫（如果使用分区适应分配，则可以为每一种大小分级维护一个链表）。当回收线程完成清扫任务后，其会将清扫所得的空闲内存块合并到全局空闲内存块链表中。该策略存在的另一个问题是，在赋值器线程耗尽其本地内存块之后，如果全局工作池中不存在更多内存块，线程应如何进行下一步动作。一种策略是从其他线程窃取内存块，但如此一来，获取下一个待清扫内存块的操作将不得不引入同步操作。尽管如此，这一开销仍然是值得付出的，因为相对于从内存块中分配一个槽的操作而言，获取新内存块进行清扫的频率通常较低，与此同时，线程在获取待清扫内存块时通常很少发生竞争。

14.8 并行整理

并行标记-整理算法所涉及的问题与本章前面所描述的其他算法基本类似。回收器首先对存活对象进行并行标记，然后再进行并行移动。并行滑动整理的实现在某些方面要比并行复制更为简单，特别是在堆空间连续的情况下。例如，当完成所有存活对象的标记后，存活对象的移动目标地址就已经确定，因此线程之间的竞争仅可能对性能造成影响，而不会影响到回收的正确性。标记阶段完成后，所有的整理式回收器都需要经过两个或更多阶段来确定存活对象的转发地址、更新引用、移动对象。正如我们在第 3 章所看到的，不同算法可能会以不同的顺序来完成这些任务，某些算法还可能在一次堆遍历过程中完成两项任务。

Crammond [1988] 为 Parlog（一种并发逻辑程序语言）实现了一种位置感知（location-aware）回收器。在逻辑程序语言中保持对象在堆中的顺序存在诸多优势，特别地，如果我们使用顺序分配策略，并在进行垃圾回收时保持对象的分配顺序，那么当程序的执行回到某一“选择点”时，在该选择点之后分配的对象全部都可以简单丢弃。滑动整理即可保持对象的分配顺序。Crammond 的回收器实现了 Morris [1978] 引线回收器（参见 3.3 节）的并行化，本节我们仅介绍该算法的并行部分。Crammond 将堆划分为多个区域，同时将每个区域与指定的处理器绑定以减少开销。在回收过程中，当处理器发现位于自身区域的对象时，会直接对其进行标记并增加本地计数器的值。一旦遇到“远程”对象，处理器会将该对象的引用压入到其所对应处理器的间接引用栈，同时增加全局计数器的值（该计数器用于结束检测）。远程处理器将负责该对象的处理并减少全局计数器的值。间接引用栈为单读多写数据结构，因而在整个算法中，只有往间接引用栈中压入远程对象时才需使用同步操作（即加锁）。Crammond 发现，通常只有不到 1% 的存活对象会被压入间接引用栈。

Flood 等 [2001] 使用并行标记-整理算法来管理其 Java 虚拟机的年老代。在并行标记阶段结束之后，回收周期还需经历三个阶段才能结束：①计算转发地址；②更新引用；③移动对象。其算法的亮点在于，整理过程的不同阶段会使用不同的负载均衡策略。单线程整理算法通常会将所有存活对象滑动到堆的一端，如果多线程并行移动对象，则必须确保每个线程不会覆盖其他线程尚未移动的存活对象。基于这一原因，Flood 等并不采用将所有存活对象整理到堆的一端的策略，而是将堆空间划分为多个区域，每个线程负责一个区域中对象的

整理。每个线程仅需将其所负责区域内的存活对象滑动到该区域的一端则可。为减少这一分区策略可能带来的（有限的）内存碎片，整理奇数编号区域的线程会与整理偶数编号区域的线程使用不同的滑动方向（参见图 14.8）。

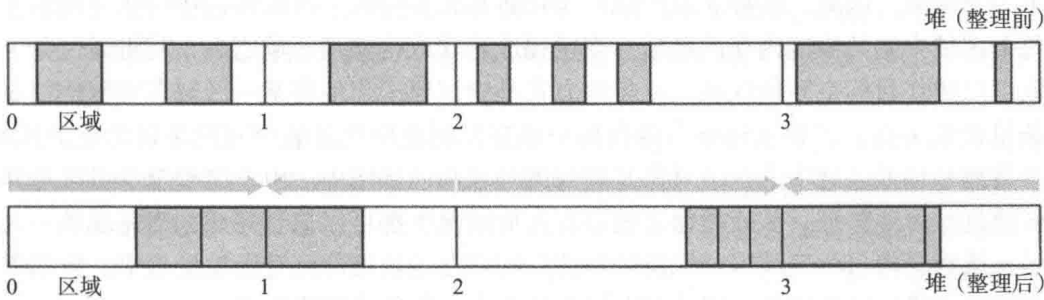


图 14.8 Flood 等 [2001] 将堆划分为多个区域，每个线程负责一个区域的整理。负责相邻区域整理的线程使用不同的滑动方向（如灰色箭头所示）

在整理过程的第一阶段，每个线程先在存活对象的头部写入转发地址，即对象在回收完成之后的新地址。该阶段中，Flood 等将堆空间进行较细粒度的划分，目的是为了实现在更好的负载均衡。他们将堆空间划分为 M 个大致相等的、依照对象对齐（object-aligned）的单元（unit），同时他们发现，在 8 路 UltraSPARC 服务器上，当单元的数量达到回收线程数量的 4 倍（即 $M = 4N$ ）时表现最佳。各回收线程通过竞争的方式来获取单元，然后计算其中存活对象的总大小，同时为了提升后续处理过程，线程还会将相邻垃圾对象合并为单个“准”对象。在计算出每个单元中存活对象的总大小之后，回收器会重新将堆空间划分为 N 个大小不等的区域，并确保每个区域所包含的存活对象数据量大致相等。这些区域会依照上一阶段所划分的单元进行对齐。该阶段同时还会计算每个单元中首个对象的目标地址，这一过程同时需要考虑每个单元中存活对象的滑动方向。计算完成后，每个回收线程将通过竞争的方式来获取单元，并为其中的每个存活对象写入转发地址。

在整理的第二阶段，回收器需要把存活对象的引用更新到其在整理完成后的新地址。与传统的整理式算法相同，这一过程仍需要扫描赋值器线程栈、与存活对象位于同一空间以及不同空间（例如更老的分代）的引用，此时有多种负载均衡策略可供选择。Flood 等再次使用单元划分策略来扫描待整理空间（即年老代），但其对年轻代的扫描则仅使用单个线程。在整理的第三阶段，每个线程将负责一块区域中对象的移动，由于每个区域中存活对象的总量大致相等，因而各线程的工作负载也大致均衡。

[300]

Flood 等的并行整理算法存在两个缺陷。首先，算法需要 3 次堆遍历过程，而第 3 章所介绍的其他算法则只需要两次甚至更少。其次，该算法最终只能将堆整理成 N 个存活对象带，以及 $\lceil (N + 1) / 2 \rceil$ 个用于分配的间隙，而不能把所有存活对象整理到堆的一端。每个存活对象带中的对象将紧密排布，意味着其中可能被浪费的空间仅可能是由对象对齐要求造成的。但是，如果回收线程的数量较多（即划分的区域较多），则很可能导致赋值器无法分配占用空间很大的对象。

Abuaiadh 等 [2004] 通过计算而非存储转发地址的方式解决了 Flood 等算法中的第一个问题，即：他们使用偏移向量来记录堆内每个小内存块中第一个存活对象的新地址，同时借助于标记位图来计算该内存块中其他存活对象的新地址，如第 3.4 节所述。为解决第二个问题，他们将堆空间划分为数个粒度较细的、大致相等的较大区域，他们建议区域的数量可以

是处理器数量的 16 倍，且确保每个区域的大小至少为 4MB。回收器将顺次对每个区域进行整理。回收线程将通过原子增加全局区域编号（或者指针）的方式获取内存区域。如果获取成功，线程将对该区域进行整理；否则意味着其他线程已经占有该区域，该线程将重新尝试获取下一个区域。因此，线程获取区域的操作将是无等待的。回收器通过一张空闲地址表来记录每个区域中起始空闲内存的地址。当线程成功获取待整理区域之后，其将通过竞争的方式获取可以用作目标空间的区域，竞争的方式是尝试原子化地将某一区域在空闲地址表中的对应条目设置为空。线程永远不会将存活对象复制到空闲地址表中对应条目为空的区域中，也不会将编号较低区域中的存活对象复制到编号较高的区域中。由于线程至少可以对其所处理的区域进行原地整理，从而确保了线程在任何情况下都可以执行完成。当完成某一区域的整理后，线程会将目标区域以及来源区域的空间内存地址更新到空闲地址表中，但如果目标区域在整理完成后已被填满，则其在空闲地址表中的条目将保持为空。

Abuaiadh 等提出了两种移动对象的方式。第一种方式是以单独的存活对象为单位来进行移动，正如前面章节所述。该方案整理效果最佳，内存碎片最少。需要注意的是，由于内存块中每个对象的移动都需要依赖该内存块在偏移向量中的值，所以回收线程必须确保同一内存块中的对象不会被隔离到不同的目标区域中。除此之外他们还提出第二种整理策略，即通过牺牲整理质量的方式来换取整理时间，该策略会一次性移动整个内存块（他们使用 256 字节的内存块），如图 14.9 所示。对象通常成簇诞生，成批死亡，因而在顺序分配策略下，存活对象与死亡对象在堆空间内也会集中出现。他们发现，这一策略可以将整理耗时减少一半，其所造成的空间浪费却几乎微不足道。但在最差情况下，该策略有可能无法回收任何空闲内存。

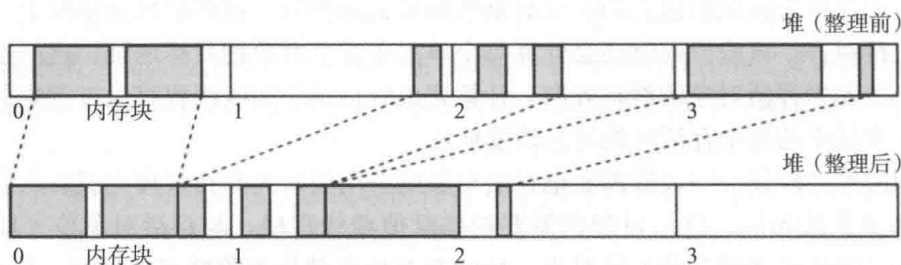


图 14.9 以内存块为单位的整理。Abuaiadh 等 [2004] 以内存块而非对象为单位进行滑动整理，每个内存块内的空闲内存将无法被“挤出”

这种计算而非存储转发地址的策略也在 Compressor 回收器 [Kermayn and Petrank, 2006] 中得到应用，但其与 Abuaiadh 等 [2004] 的策略有所不同。首先，在第二阶段对标记位图的遍历过程中，回收器在计算偏移向量^①之外还需计算首对象向量（first-object vector）。首对象向量以整理完成之后存活对象所占用内存页的编号作为索引，其每个槽所对应的值是未来将会移动到对应页中的首个对象在执行整理之前的地址。整理过程从更新根开始执行（该过程需要依赖标记向量与偏移向量）。

Compressor 回收器的第二个不同之处在于，每个线程将通过首对象表来竞争目标页，竞争获胜的线程将把一个新的物理内存页映射为虚拟内存页，然后根据目标页在首对象向量中所记录的地址进行对象复制，复制过程需要同时依赖偏移向量与标记向量。这种将存活对象迁移到新内存页的策略允许 Compressor 回收器使用多个并行回收线程，且整理过程能够

^① 其偏移向量中的每个条目对应 512 字节，其内存块同样也比 Abuaiadh 等 [2004] 的要大。

遵循滑动顺序（见第3章）。我们很容易将 Compressor 回收器误认为是复制式而非整理式回收器，但它确实是一种遵循滑动顺序的标记-整理回收器：尽管该回收器在管理来源空间页与目标空间页时也需要付出一定的空间开销，但这一开销通常只是每个回收线程的一个内存页，相比之下，传统的半区复制回收器则需要两倍的堆空间。Compressor 回收器的技巧在于，它不仅映射新的目标空间页，还可以在完成来源空间某一页中全部存活对象的迁移后解除其映射。

这一设计策略最大限度地减少了各整理线程之间的同步开销，即：只有当线程从首对象向量中为其目标页获取首对象地址时，才需要使用同步操作，同时由于线程在竞争失败之后无需进行重试，所以这一过程也是无等待的（某一线程竞争失败意味着其他线程竞争胜出，因而该线程只需尝试获取下一个条目）。算法的结束检测也十分简单，每个线程只需在到达首地址向量的末尾后直接退出。一个值得注意的细节是跨页对象的处理。在万物静止算法中，我们可以简单地将跨页对象与其最终将被移动到的第一个页相关联。但是在并发回收算法中这一方案将存在问题（我们将在 17.7 节进行详细讨论），此时我们必须精确复制属于某一目标空间页的数据，即：如果某一对象只有前半落入该页，我们则只能复制其前半部分；类似地，如果某一对象只有后半落入该页，我们则只能复制其后半部分。

14.9 需要考虑的问题

14.9.1 术语

在对并行垃圾回收的早期研究中，相关术语的使用通常较为混乱，20 世纪的论文中通常会将“并行”、“并发”甚至“即时”这些术语等价。但幸运的是，学者们已经在 2000 年左右对这些术语的含义达成共识：并行回收器仅指基于多个回收线程并行执行的回收器，在回收过程中，赋值器线程既可能处于挂起状态，也可能继续执行。毋庸置疑，在底层平台支持的情况下，我们应当尽量使用并行回收，正如赋值器尽量多地使用并行处理器资源那样。

[302]

14.9.2 并行回收是否值得

首先我们需要考虑的问题是 Amdahl 定律[⊖]的限制，即是否有足够多的工作可以并行。无法并行处理的案例十分普遍，例如对单链表进行追踪。事实证明，实际的应用程序中通常会存在多种类型的数据结构，且它们通常具有较高的潜在并行加速率 [Siebert, 2008]。在追踪过程之外，垃圾回收过程的其他动作也赋予并行硬件更多的用武之地，例如清扫过程与整理过程显然是可以并行化的（尽管并行整理处理起来稍微复杂一些）。即使是在追踪阶段，对线程栈以及记忆集的并行扫描也只需要引入很少的同步开销，而并行追踪则需要仔细设计各线程的工作列表，这不仅是为了尽量减少同步开销，而且是为了尽量高效地利用并行硬件资源。

14.9.3 负载均衡策略

高效的并行回收算法需要在处理器的负载均衡与线程同步开销之间进行平衡。负载均衡

⊖ Amdahl 定律指出，并行处理对应用程序的加速效应受限于程序中可以并行处理的工作占全部工作的比例。设 s 为程序中必须串行处理的工作在单个处理器上的执行时间， p 为可以由 n 个处理器并行处理的工作在单个处理器上串行执行需要花费的时间，则并行加速比为 $1 / (s + p / n)$ 。

的目的在于避免出现部分处理器空闲而其他处理器忙于处理所有工作的情况。对内存等其他资源进行均衡分配也十分重要。同步的目的不仅在于保护回收器的工作列表，同时还是为了确保从堆中所分配的数据结构的完整性。例如，两个线程同时操作同一个标记栈指针可能会导致元素的丢失，而两个线程同时复制同一个对象则可能导致对象图的拓扑结构发生错误的变化。另外，最细粒度的负载均衡策略通常会伴随极高的同步开销。

为此，负载均衡的解决方案一般是为每个回收线程分配一定量的工作，且线程在处理这些工作时无需与其他线程进行额外同步，因此如何在线程之间进行工作分配便成为关键。最直接且同步开销最小的工作划分方案是对回收工作进行静态划分，执行这一划分的时机可以是编译期，也可是程序启动或回收过程开始时。在静态工作划分策略下，线程仅在执行结束检测时才需进行同步，但该策略的缺陷在于线程之间通常无法实现较好的负载均衡。另一种负载均衡策略是将回收工作进行较细粒度的划分，每个线程通过竞争的方式获取工作，同时将新产生的工作释放到全局工作池中。该策略可以达到较好的负载均衡效果，但其可能会引入过多的同步开销。在这两种极端策略之外，我们也可在回收过程的不同阶段使用不同的负载均衡策略。例如，某一阶段所获取的信息（通常是标记阶段）通常可以指导后续阶段各线程之间的工作划分，Flood 等 [2001] 的回收器便是使用这一策略的绝佳案例。

14.9.4 并行追踪

追踪堆的过程通常会涉及两个方面：一是现有工作的处理（即标记 / 复制对象），二是新工作的生成（即处理对象尚未得到追踪的子节点）。该阶段通常会使用栈或者队列等数据结构来记录待处理工作。单个共享数据结构通常会引发较大的同步开销，因而回收线程通常会配备本地数据结构。但出于负载均衡的目的，我们仍需要引入一些额外机制来实现线程之间的工作迁移。首先需要确定的是使用哪种机制进行工作迁移。本章介绍了多种工作迁移策略。支持工作窃取的数据结构允许线程之间安全地进行工作传递，该策略通常会在确保常态操作（即追踪过程中的栈压入和弹出操作）尽量廉价（即不使用同步操作）的前提下支持非常态操作（即线程之间工作的安全传递）。Endo 等 [1997] 为每个线程提供了本地栈以及本地可窃取工作队列，而在 Flood 等 [2001] 的方案中，每个线程仅需使用一个同时支持追踪与工作窃取的双端队列。灰色工作包策略使用一个全局工作池来保存需要处理的工作缓冲区 [Thomas 等, 1998 ; Ossia 等, 2002]，每个回收线程会以竞争的方式获取工作包并进行处理，同时线程新产生的工作也将以新工作包的形式释放到全局工作池中。Cheng 和 Blleloch [2001] 将追踪过程拆分为多个阶段（他们称之为“工作空间”），在最简单的情况下，所有线程要么都位于压入工作空间，要么都位于弹出工作空间。不论线程位于哪个工作空间，所有线程在相同时刻都将尝试沿着相同的方向移动栈指针，此时使用 FetchAndAdd 原子操作便可满足要求。还有一些回收器会在任意两个追踪线程之间建立单读单写通道，从而无需依赖任何原子操作 [Wu and Li, 2007 ; Oancea 等, 2009]。

另外还需考虑的问题是，线程之间应当传递多少工作，以及何时进行传递。研究者们提出了多种不同的解决方案。工作传递的最小单元可以是栈中的单个元素，但如果传递单元过小，则可能增大线程之间的传递流量。在 Sierbert [2010] 并行、并发以及实时回收器中，回收器会保留一个处理器，该处理器在初始情况不会分配到任何工作，其所有工作均是从其他线程窃取而来。但该方案似乎只有在各线程不会在相同时间完成回收工作时才能显示出其价值（因为该回收器允许赋值器与回收器并发执行，因而不存在处理器资源浪费问题）。通用

的解决方案是将线程之间传递工作的单元设置为某一适中的值。固定大小的灰色工作包天然满足这一要求，除此之外还有其他一些策略，例如传递线程标记栈中工作量的一半。如果标记栈的大小固定，则某些实现机制必须能够处理栈溢出问题。灰色工作包也可天然解决这一问题：当输出工作包被填满后，线程会将其释放到全局工作池中，并从中获取一个新的空工作包。Flood 等 [2001] 将溢出的对象通过引线的方式与 Java 类对象链接，该策略仅会为每个类引入较小的、固定大小的空间开销。大数组会给线程之间的负载均衡带来一定麻烦，一种解决方案是将较大的、逻辑连续的对象拆分为链式数据结构，该方案在实时系统中得到广泛应用。另一种解决方案是将大数组划分为多个分段压入标记栈，从而可以将整个数组的扫描划分为多个步骤。

上述各种负载均衡策略均是以处理器为中心的，即算法侧重于管理每个线程（处理器）的本地工作列表。另一类负载均衡策略则是以内存为中心，即算法主要考虑对象在内存空间的位置分布。这一负载均衡策略在非一致内存架构中将格外重要，因为处理器访问远端内存的开销要比访问本地内存大得多。以内存为中心的负载均衡策略在并行复制式回收器中应用广泛，特别是在以 Cheney 队列作为工作列表的回收器中 [Imai and Tick, 1993 ; Siegart and Hirzel, 2006]。此类负载均衡策略需要考虑的主要问题是：①内存块的大小（即线程的最小处理单元）；②线程应当获取哪个内存块进行处理，同时又应当将那个内存块释放到全局工作池中；③对象被哪个线程所“拥有”。在确定内存块大小时有两方面因素需要考虑。第一，所有移动式回收器都应当为每个线程配备私有分配缓冲区，目的是为了进行速度较快的顺序分配。为避免给全局内存块分配器造成压力，这些用于线程本地分配缓冲区的内存块应当拥有较大的空间，但对于使用复制策略的并行回收器而言，较大的内存块不利于线程之间的负载均衡。因此对于 Cheney 式回收器而言，用作线程本地分配缓冲区的内存块应当被划分为更小的内存块。第二，如何选择下一个要处理的对象将同时影响回收器和赋值器的局部性（参见 4.2 节）。较好的选择应当是在用于分配的内存块中选择一个对象进行处理，而将处理过程中间生成的、未扫描的或尚未完成扫描的内存块释放到全局工作池中。如果是在扫描完一个内存块之后再选择下一个内存块，将有助于提升回收器的局部性；而如果在完成一个对象的扫描之后立即选择下一个待扫描内存块，则有助于提升赋值器的局部性，因为这一策略将使回收线程以类似于深度优先的顺序（即层次分解顺序）进行追踪。最后，还需考虑的问题是某一对象最可能被哪个处理器访问。Oancea 等 [2009] 使用“支配线程”这一概念来决定每个对象应当被哪个处理器复制（同时也决定了对象将被复制到哪个空间）。

[304]

14.9.5 低级同步

我们知道，对回收器数据结构的修改通常需要使用同步操作，而对单个对象的更新也可能需要使用同步操作。原则上讲，标记过程本身属于幂等操作，也就是说，即使同一对象被标记多次也不会出错，但如果回收器使用位图标记，则必须确保标记操作的原子性。由于现代处理器指令集通常不会提供设置一个字或字节中的某一位的原子操作，所以设置标记位的过程通常需要对整个标记字节进行原子化的设置，如果设置失败，则需进行重试，直到设置成功为止。如果将标记位保存在对象头部，或者使用字节图进行标记，则标记过程无需引入任何同步操作。

复制式回收器在任何情况下都不应当多次“标记”（即复制）同一个对象，这可能导致对象图拓扑结构的变化，同时还可能严重破坏可变对象的一致性。并行复制的一个基本要求

是，线程复制对象、设置转发地址的操作在其他回收线程看来应当是单个不可分割操作，这一要求的实现归根结底在于转发地址的处理方式。研究者们提出了多种解决方案。在某些回收器中，回收线程先在对象的转发地址槽中原子化地写入某个意味着“繁忙”的值，然后再复制对象并写入转发地址；期间，如果其他线程读取到这一“繁忙”值，则必须进行自旋直到读取到真正的转发地址。如果线程在尝试原子化地写入“繁忙”值之前先对转发地址槽进行判断，则可以进一步减少同步开销。另一种策略是，如果线程发现某一对象尚不存在转发地址，则先复制对象，然后再尝试原子化地写入转发地址，如果写入失败，则需将复制操作撤销。这一策略的效率取决于线程在设置转发地址时的冲突概率。

对于内存一致性模型较弱的硬件平台，确保某些操作能够以合适的顺序被其他处理器感知也十分重要，这需要编译器在合适的位置插入内存屏障。诸如 CompareAndSwap 等原子操作通常都可以扮演内存屏障的角色，但是在许多场景下一些更弱的指令就已经足够。在选择具体的并行算法时，决定在何处放置屏障的复杂性、屏障的执行次数以及屏障操作的开销，都是必须予以考虑的重要方面。以牺牲少许性能为代价来换取编程方式的简化（从而确保代码的正确性），通常来说是值得的。

14.9.6 并行清扫与并行整理

清扫与整理阶段通常会在堆中进行线性遍历（整理过程通常需要多次遍历），因而这两类操作比较适合进行并行处理。最简单的负载均衡策略可能是将堆划分为与处理器数量相等的多个分区，但如果各分区内的存活对象数量差异较大，则很容易产生负载不均衡。针对这一问题，我们可以使用分区中存活对象的数量来近似该分区的处理工作量，这一信息通常可以在标记阶段计算得出，回收器可以据此将堆划分为大小不等（但依照对象对齐）的分区，并确保每个分区的处理工作量大致相等。

这一策略有效的前提是每个分区可以独立于其他分区进行处理，但如果对某一分区的处理可能会破坏其他分区所依赖的信息，则算法的处理便可能出现問題。例如，滑动整理回收器不能以任意顺序来移动对象，否则可能会覆盖尚未移动的存活对象，此时回收器可能需要依照地址顺序来处理各分区。一种解决方案是将堆划分为更细粒度的分区，每个整理线程以竞争的方式获取来源分区以及目标分区。

14.9.7 结束检测

回收器应当能够正确地检测出回收过程中每个阶段的结束。多线程的并行增大了结束检测的复杂度，其根本原因在于，在某一线程判定当前阶段是否结束的同时，其他线程可能还会生成更多的工作。不幸的是，正确地实现结束检测算法并非易事。一种（正确的）解决方案是指定一个线程来进行结束检测，而其他线程则需要原子化地设置自身的某一旗标以反映其是否繁忙，在实现过程中还需特别注意旗标的设置、清除与线程处理工作之间的顺序关系，在松散内存一致性模型的平台下还需在适当的位置插入内存屏障。使用全局工作池的系统允许更加简单的结束检测机制，且允许任意数量的线程同时进行结束检测。例如，灰色工作包系统允许线程计算全局工作池中工作包的数量，如果全部工作包都位于全局工作池中，且其全部为空，则当前阶段结束。

并发垃圾回收

并发垃圾回收的设计初衷是降低单处理器环境下的垃圾回收停顿时间。一些早期的文献会将“并发”(concurrent)、“并行”(parallel)、“即时”(on-the-fly)、“实时”(real-time)这几个术语等价或者混淆。第 14 章已经描述了“并行”这一术语在当前业界的普遍用法，这里我们将对其余几个术语进行定义。目前为止，我们均假定在垃圾回收处理过程中赋值器始终处于挂起状态，并且只有当所有回收线程处理结束之后，赋值器才能恢复执行。图 14.1 以条带图的方式介绍了不同类型的万物静止式回收：时间前进方向为从左到右，白色条带代表赋值器的执行时间，其他颜色条带代表回收器的执行时间，其中灰色条带代表一个垃圾回收周期中回收器的行为，而黑色条带代表下一个回收周期。

在第 14 章中，我们介绍了一种通过多处理器来减少回收停顿时间的策略，即在赋值器线程挂起的同时，使用多个回收线程并行处理一个回收周期内的工作任务，如图 14.1c 所示。

在单处理器环境下降低回收停顿时间的另一种策略是让赋值器与回收器交替执行，如图 15.1a 所示。该策略中的一个回收周期会被分割为多个细粒度的回收增量，因而称其为增量回收 (incremental collection)。但是，增量回收却不像图 15.1a 直接反映出来的那样简单，因为在赋值器看来，回收周期不再具有原子性，所以在相邻两个回收增量之间，对象的可达性可能会发生变化。因此，增量回收器必须通过某种方式来跟踪对象图中可达对象的变化，甚至可能需要重新扫描可达性发生变化的对象或者域。为解决这一问题，研究者们提出了多种不同的策略。

尽管交替执行已经引出了赋值器和回收器并发执行的问题，但在增量回收中，赋值器与回收器永远不会并行，即在每个回收增量中赋值器均处于挂起状态。我们可以将这一特征推广到多处理器环境下，即确保在每个回收增量中所有赋值器线程均处于挂起状态，如图 15.1b 所示；与此同时，每个回收增量的处理过程也可并行化，如图 15.1c 所示。

从概念上讲，我们可以简单地将单处理器环境下赋值器和回收器的交替执行策略推广到多处理器环境下，即（多）赋值器与回收器并行执行，但这里的主要困难之处在于，如何才能确保赋值器和回收器在对象图的拓扑结构方面保持相同的认知（对象可达性只是其中的一方面），这便需要在两者之间引入适当的同步。例如在回收器工作过程中，如果赋值器尝试更新某个仅完成部分扫描或者部分复制的对象，或者与回收器同时访问某一元数据时，都可能导致不一致的出现。

赋值器与回收器之间的同步程度 (degree) 和同步粒度 (granularity) 影响着应用程序的吞吐量（即包括两者在内的整体执行时间）。在回收过程的某些阶段，同步是一种远比其他方法简单的处理方式。主体并发回收 (mostly-concurrent collection) 在每个回收周期中会令所有赋值器线程挂起一小段时间（通常是在回收周期的开始阶段），回收器将在这段时间内完成线程栈扫描等操作，而在其他时间，赋值器则可以与回收器同时执行，从而减少了两者之间的同步开销。主体并发回收既可以是整体式的（见图 15.1d），也可以是增量式的（见图 15.1e），（期望停顿时间较短的）万物静止阶段可以确保所有赋值器线程同时感知到回收过程

的开始。

如果彻底消除回收过程中的万物静止阶段，则回收过程就成为纯粹的并发即时回收 (on-the-fly collection)，此时不论在任何阶段，回收器都将与赋值器并行（见图 15.1f），即时回收过程也可以是增量式的（见图 15.1g）。白色条带中的竖线意味着在每个回收周期的开始阶段，每个赋值器线程依然可能需要与回收器进行同步，但整个系统将不存在全局的万物静止阶段[⊖]。

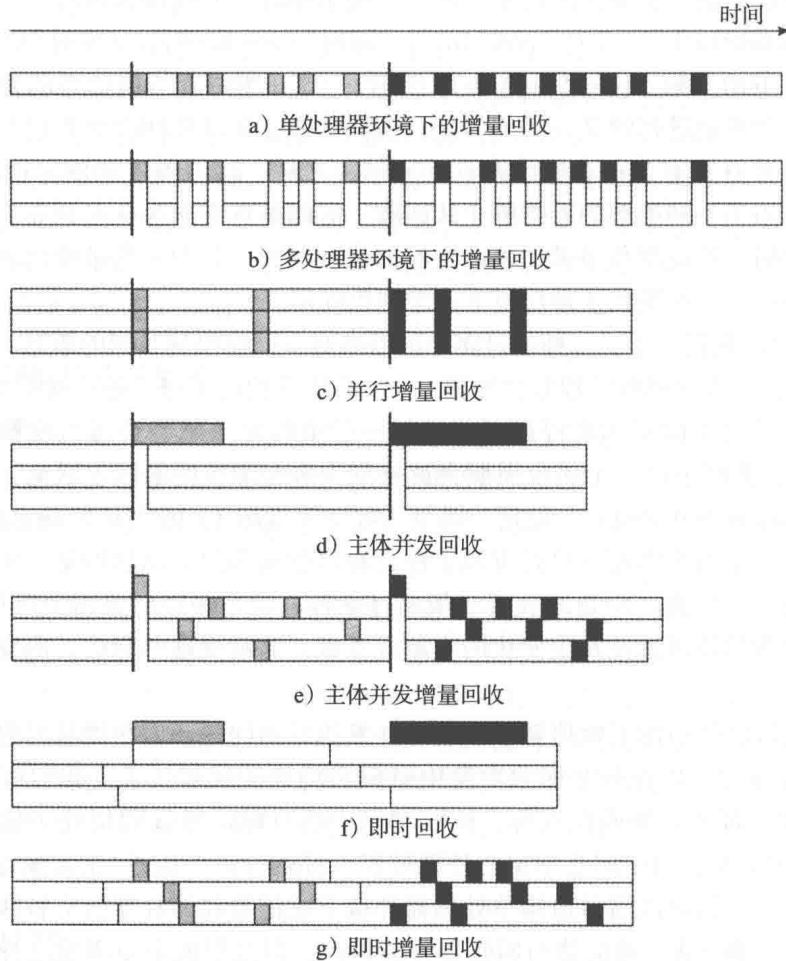


图 15.1 增量回收与并发回收。每个条带代表一个处理器的执行过程，带颜色的区域代表不同的垃圾回收周期

15.1 并发回收的正确性

正确的并发回收算法必须满足两个条件：

- 安全性要求回收器至少必须保留所有可达对象；
- 存活性要求一个回收周期最终必须能够结束。

控制赋值器和回收器交替执行的能力是确保并发回收器正确执行的必要条件。在并发回收算法中，赋值器和回收器的各项操作必须是原子化的，只有这样，我们才能将它们之间的

[⊖] 并发回收在历史上曾经与即时回收等价 [Dijkstra 等, 1976, 1978; Benair, 1984]，但在当前，即时回收通常特指不会同时将所有赋值器线程挂起的垃圾回收。

交替操作看作是由单个赋值器（和单个回收器）所执行的，同时不失一般性。回收算法可以使用任意一种能够确保这些原子操作按序执行的调度算法，因此在算法的具体实现中，究竟选择何种并发控制策略便存在较大的自由度。为确保这些操作满足原子化要求，最简单的方法可能是增量式地进行垃圾回收，即令回收器与赋值器交替执行，并确保在每个回收增量运行中所有赋值器线程均处于挂起状态。当然还有一些其他的策略可以满足更细粒度的同步要求，具体实现方法可参见第 13 章。

15.1.1 三色抽象回顾

推导并发回收正确性最简单的方法是使用三色抽象，它是所有赋值器与回收器都必须遵守的不变式。所有并发回收器都必须遵守三色抽象不变式中的某些要求，除此之外，回收器还必须能够（安全地）保留所有可达对象，即使赋值器在回收过程中修改了这些对象。回顾三色抽象我们可知：

- 白色对象尚未被回收器访问到，在回收周期的开始阶段，所有对象均为白色，而当回收周期结束时，所有白色对象均为不可达对象。
- 灰色对象已经被回收器访问过，但回收器仍需对其中的一个或者多个域进行扫描（它们可能指向其他白色对象）。
- 黑色对象已经被回收器访问过，且其所有域都已被扫描过。黑色对象的任何一个指针域都不可能直接指向白色对象。黑色对象永远不会被回收器重新扫描，除非它的颜色发生变化。

我们可以把回收过程地看作是回收器以灰色对象为“波面”（wavefront）并不断向前推进的过程，这一波面同时也是黑色对象（在某一时刻可达，且已经被回收器扫描过）和白色对象（尚未被回收器访问过）的边界。如果没有赋值器并发修改操作的干扰，回收周期的顺利结束将不存在任何问题。但并发回收的核心问题在于，由于赋值器会在回收过程中并发更新对象图，所以赋值器和回收器可能会在对象图的拓扑结构方面产生不同的认知，此时再以灰色波面作为黑色对象与白色对象的边界便不再适合。

回顾第 1 章所定义的赋值器 Write 操作，我们可以对其进行适当修改，即在域中写入新值之前先读取其中原有的值：

```
atomic Write(src, i, new):  
    old ← src[i]  
    src[i] ← new
```

Write 操作在对象 src 的 src[i] 域中插入了指针 $\text{src} \rightarrow \text{new}$ ，其副作用是删除了同一域中原有的指针 $\text{src} \rightarrow \text{old}$ 。atomic 关键字意味着 old 和 new 指针将在一瞬间完成交换，期间不会穿插其他额外的赋值器/回收器操作。对于大多数硬件而言，存储操作天然就是原子化的，因而 Write 操作无需额外引入其他显式同步操作。

当赋值器与回收器并发执行时，如果赋值器所修改的对象位于追踪波面之前——包括灰色对象（回收器尚未对其内部指针域进行扫描）和白色对象（回收器尚未访问过这些对象）——则回收的正确性不会受到影响，因为回收器终究会在某一时刻访问这些对象（如果这些对象依然可达）。即使赋值器所修改的对象位于追踪波面之后——即黑色对象（回收器已完成对其内部域的扫描）——只要其所插入或删除的指针目标对象为黑色或者灰色（即回收器已经判定该对象可达），也不会存在任何问题。但是，除此之外的其他指针更新操作极有可能导致赋值器和回收器对存活对象集合产生不一致的认知 [Wilson, 1994]，进而导致存

活对象被错误地回收。下面我们将考察一个具体案例。

15.1.2 对象丢失问题

图 15.2 展示了两种将白色指针插入到追踪波面之后的场景。图 15.2a 所描述的情况是：对于原本从灰色对象直接可达的对象，如果赋值器先将其引用插入到波面之后，然后再从灰色对象中删除其引用，则可能导致该对象“丢失”。在初始状态下，堆中存在一个黑色对象 x 以及一个灰色对象 y ，其中后者已被标记为从根可达，还有一个白色对象 z ，可直接从 y 可达。在操作 D1 中，赋值器从灰色对象 y 中加载指针 a ，并将其插入到对象 x 中，最终得到从 x 指向 z 的指针 b 。在操作 D2 中，赋值器将尚未扫描过的指针 a 从 y 中删除，而该指针却是唯一一个从灰色对象指向 z 的指针。在操作 D3 中，回收器完成对象 y 的扫描并将其着为黑色，标记阶段就此结束。而在清扫阶段，即使白色对象 z 依然经由指针 b 可达，回收器仍会错误地将其回收。

图 15.2b 所描述的情况是：对于经由一条传递指针链从灰色对象可达的对象，如果赋值器先将其引用插入到追踪波面之后，然后再删除该指针链上的某个指针，也可能导致该对象“丢失”。与 15.2a 所示的情况不同，此时赋值器并未删除任何直接指向已丢失对象的指针。在初始状态下，堆中存在一个黑色对象 p 以及一个灰色对象 q ，其中后者已被标记为从根可达。白色对象 r 直接从 q 可达，而白色对象 s 则需要经由 r 从 q 可达。在操作 T1 中，赋值器从白色对象 r 中加载指针 d 并将其插入对象 p 中，最终得到从 p 指向 s 的指针 e 。在操作 T2 中，赋值器删除指向 r 的指针 c ，该操作导致从灰色对象 q 到 s 的唯一一条指针链被破坏。在操作 T3 中，回收器完成对象 q 的扫描并将其置为黑色，标记阶段就此结束。而在清扫阶段，即使白色对象 s 依然经由指针 e 可达，回收器仍会错误地将其回收。

[310]

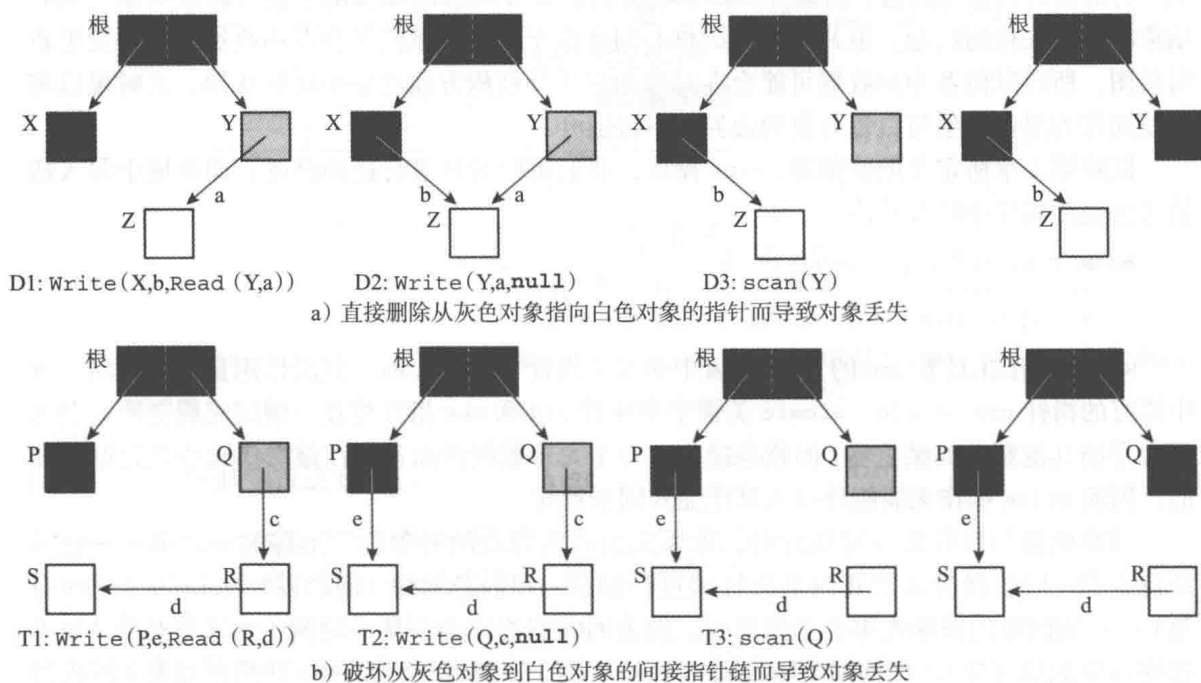


图 15.2 对象丢失问题。如果赋值器操作破坏了白色对象与灰色对象之间的可达关系，则可能导致回收器错误地回收可达对象

Wilson[1994]发现,在追踪过程中,只有当如下两个条件同时满足时才会出现对象丢失问题:

- 条件 1: 赋值器将某一指向白色对象的引用写入黑色对象;
- 条件 2: 从灰色对象出发,最终到达该白色对象的所有路径都被赋值器破坏。

当赋值器将白色指针(即指向白色对象的指针)插入到黑色对象时,如果回收器无法通过其他指针遍历到该对象,则会出现对象丢失问题。也就是说,即使该白色对象(从黑色对象)可达(即条件 1),但由于回收器不会重新扫描黑色对象,所以其永远不会发现这一事实。因此,如果回收器最终能够遍历到该白色对象,唯一可能的途径便是从某个已经遍历到但尚未完成扫描的对象(即灰色对象)出发,经由一条尚未访问过的路径(即由白色指针构成的指针链)并最终到达该对象,但条件 2 却致使堆中不存在这样一条路径。

15.1.3 强三色不变式与弱三色不变式

为确保回收器不会错误地回收存活对象,我们必须确保导致对象丢失的两个条件不会同时出现。要确保回收器不会漏掉任何可达对象,其必须能够找出所有被黑色对象引用的白色对象。为此,只要能确保所有被黑色对象引用的白色对象不被删除,它们将不会被回收器遗漏,此时这些白色对象将处于灰色保护(grey protected)状态。对于直接从灰色对象可达,或者经由一条指针链从灰色对象可达的白色对象,其天然不存在对象丢失问题。因此,可能导致对象丢失问题产生的条件 2 将不可能出现,因而回收器只需满足:

- 弱三色不变式(weak tricolour invariant): 所有被黑色对象引用的白色对象都处于灰色保护状态(即直接或间接从灰色对象可达)。

非复制式回收器天然存在一个优势,即一旦某个指针的目标对象变成灰色或者黑色,则该指针将自动成为灰色/黑色指针^①。因此,黑色对象中的白色指针将不会成为问题,因为它们所指向的、处于灰色保护状态的白色对象最终都将被回收器着色,即在追踪过程结束之前,黑色对象中的所有白色指针最终都将变成黑色。

并发复制式回收器则稍加复杂,因为在回收结束时,每个存活对象将存在两个副本(来源空间中为白色副本,目标空间中为黑色副本),此时回收器将把白色副本与其他垃圾一起回收。从概念上讲,回收器永远不会重复访问黑色对象,因此,并发复制式回收器要满足正确性要求,其必须确保永远不会在目标空间的黑色对象中写入白色来源空间指针(即指向来源空间中白色对象的指针),否则在回收结束时,目标空间的黑色对象中将存在指向(已被回收的)来源空间的悬挂指针。为此回收器必须满足:

- 强三色不变式(strong tricolour invariant): 不存在从黑色对象指向白色对象的指针。

满足强三色不变式要求的回收器必然满足弱三色不变式,但反之不然。赋值器在黑色对象中插入白色指针是造成对象丢失问题的根源,因而只要我们能避免这一情况的出现,便可解决对象丢失问题。强三色不变式不仅适用于复制式回收器,同时也适用于非复制式回收器。

在图 15.2 所示的两种情况中,赋值器均会在黑色对象中写入某一白色对象的引用(D1/T1),从而打破了强三色不变式;之后赋值器又破坏了所有从灰色到达该白色对象的路径(D2/T2),进而又打破了弱三色不变式。这两步操作最终导致(可达)黑色对象包含了指向

① 因为在非复制式回收器中,回收器不会修改指向存活对象的指针。——译者注

(可能是垃圾的)白色对象的指针,进而破坏了回收器的正确性。要解决这一问题,赋值器必须在写入指向白色对象的指针时(D1/T1),或者在删除可达对象的可达路径时(D2/T2)引入额外的操作。

15.1.4 回收精度

对于以不同策略来满足安全性与存活性要求的不同回收算法,它们在回收的精度(precision,由回收结束时剩余对象的集合决定)、效率(efficiency,意味着吞吐量)、原子化程度(atomicity,意味着并发程度)等方面均会存在不同表现。不同的回收精度意味着在回收结束时,剩余对象集合将是存活对象集合的一个超集,进而影响垃圾回收的及时性。万物静止式回收器可以达到最大化的回收精度(即所有不可达对象都将得到回收),但其却丧失了任何与赋值器并发执行的可能性。细粒度的原子性可以提高回收器与赋值器之间的并发程度,但其代价是导致更多的不可达对象残留在堆中,且关键操作的原子化也会产生额外的开销。确定追踪过程中最小但满足要求的临界区集合存在一定难度,Vechev等[2007]展示了如何将这一查找过程半自动化。对于在回收周期结束时依然存在的不可达对象,我们将其称为浮动垃圾,尽管它们并不会严重影响到回收器的正确性,但通常我们不希望堆中存在过多的浮动垃圾。对于并发回收器而言,浮动垃圾是否能在未来几个回收周期中得到回收,是检验回收器完整性的一个重要指标。

15.1.5 赋值器颜色

在对回收算法进行归类时,如果把回收器也看作是对象,则为赋值器根引入颜色的概念也是十分有用的。如果某一赋值器尚未被回收器扫描过(即回收器的根尚未被追踪到),或者尽管已经扫描过但仍需重新扫描,则称其为灰色赋值器。灰色赋值器的根所引用的对象既可能是白色,也可能是灰色或者黑色。黑色赋值器已经由回收器扫描过,回收器已完成其根的追踪,且不会再次对其进行扫描。强三色不变式要求黑色赋值器的根只能引用灰色或者黑色对象,但不能引用白色对象;弱三色不变式允许黑色赋值器的根引用白色对象,但前提是这些白色对象必须处于灰色保护状态。

赋值器的颜色会对回收周期的结束产生影响。从定义上讲,如果某种并发回收器允许灰色赋值器的存在,则其必须在回收结束之前重新扫描该赋值器的根,而如果重新扫描过程中发现了新的灰色或者白色对象,回收器还需对新发现的对象进行追踪,但是在新的追踪过程中,赋值器依然可能在其根中插入新的非黑色引用,因而在追踪过程结束后,回收器仍需再次扫描其根。对于允许灰色赋值器存在的算法,在最差情况下,回收器只有将所有赋值器线程挂起才能完成其根的最终扫描。

到目前为止,我们均假设系统中仅存在一个赋值器。但在即时回收算法中,由于回收器不能同时挂起所有赋值器线程并扫描其根,所以必须区别对待每个赋值器线程,此时回收器便可能同时处理灰色(尚未完成扫描)和黑色(已完成扫描)两种不同颜色的赋值器线程。另外,某些回收器还会将单个赋值器线程的根划分为已扫描分区(黑色)和未扫描分区(灰色),例如将线程栈帧顶部的已扫描区域标记为黑色,其他未扫描区域则仍为灰色。如果线程通过函数返回或者栈展开的方式进入到灰色分区,则回收器必须对新的栈顶区域进行扫描。

[313]

15.1.6 新分配对象的颜色

分配过程会导致赋值器持有新分配对象的引用。在(强/弱)三色不变式的要求下,新

分配的对象也必须被赋予适当的颜色，因此赋值器的颜色也会影响其所分配对象的颜色。新分配对象的颜色会影响其在不可达时的回收及时性：如果新分配的对象为黑色或者灰色，即使赋值器直接将其抛弃而并未将其写入堆中，该对象也不可能在当前回收周期中得到回收（因为黑色和灰色意味着可达）。灰色赋值器可以分配白色对象以避免不必要的新对象保留；而黑色赋值器则不能分配任何白色对象（强三色不变式和弱三色不变式均要求如此），唯一的例外是（在弱三色不变式下）确保赋值器将该白色对象的引用写入波面之前的某一存活对象中，只有这样才能确保回收器最终能够遍历到该对象，否则即使该白色对象被黑色对象所引用，回收器依然会错误地将其回收。需要注意的是，新分配对象在初始状态下不会包含任何外部引用，因而在分配时将其着为黑色通常是安全的。

15.1.7 基于增量更新的解决方案

在解决对象丢失问题的各种策略中，某些策略将注意力集中在图 15.2 中的 D1/T1 操作上，Wilson[1994] 将此类解决方案称为增量更新（incremental update）技术。此类解决方案会把赋值器对已知存活对象集合的增量修改通知给回收器，进而产生了需要额外（重新）扫描的对象。如果某一对象的引用被插入到追踪波面之后的黑色对象中，增量更新方案会保守地将其当作存活对象，因而即使赋值器破坏了该对象在追踪波面之前的所有可达路径，也不会产生对象丢失问题，因此增量更新技术满足强三色不变式的要求。此类解决方案使用赋值器写屏障来拦截将白色指针插入黑色对象的行为，以图 15.2a 为例，写屏障会将指针 b 的目标对象着为黑色，从而避免产生从黑色对象指向白色对象的指针。

对于黑色赋值器从堆中加载引用的操作，我们可以将其看作是在黑色对象（即赋值器本身）中插入指针的操作，因而增量更新技术需要使用读屏障来拦截将白色指针插入黑色赋值器的操作。

15.1.8 基于起始快照的解决方案

还有一些策略致力于在 D2/T2 赋值操作中解决对象丢失问题，此类解决方案会保留回收开始时刻的存活对象集合，Wilson 将其称为起始快照（snapshot-at-the-beginning）技术。当赋值器从灰色或者白色对象（即位于追踪波面之前的对象）中删除白色指针时，写屏障会将这一行为通知给回收器。如果某一指针位于追踪波面之前，基于起始快照的解决方案会保守地将其目标对象当作存活（非白色）对象，此时即使赋值器将其引用插入到追踪波面之后，也不会产生的对象丢失问题。此类解决方案可以确保在回收过程中，对于任意一个在回收开始时刻可达的对象，赋值器都不可能将其全部可达路径破坏，因而其满足弱三色不变式的要求。基于起始快照的解决方案使用写屏障来拦截赋值器从灰色或白色对象中删除灰色或白色指针的行为。

获取赋值器的快照意味着回收器需要扫描其根并将其着为黑色。我们必须在回收起始阶段完成赋值器快照的获取，并确保其不持有任何白色指针。否则，一旦赋值器持有某一白色对象的唯一引用并将其写入黑色对象，然后再抛弃该指针，则会违背弱三色不变式的要求。为黑色对象增加写屏障可以捕获这一插入操作，但如此一来，该方案将退化到强三色不变式框架下。因此，基于起始快照的解决方案将只允许黑色赋值器的存在。

314

15.2 并发回收的相关屏障技术

赋值器屏障是确保回收器正确维护强 / 弱三色不变式的基础之一。Pirinen [1998] 指出，

赋值器屏障需要依赖多种操作来应对指针的插入与删除，包括：

- 扩大 (add to) 波面：将白色对象着色 (shade) 为灰。对灰色和黑色对象再次着色不起作用。
- 推进 (advance) 波面：扫描 (scan) 对象并将其着为黑色。
- 后退 (retreat) 波面：将黑色对象回退 (revert) 到灰色。

除此之外，只有两种操作可能打破强 / 弱三色不变式，一是将某一对象回退到白色，二是不经过扫描便将某一对象着为黑色。算法 15.1 到 15.2 枚举了并发回收中的经典屏障技术。

算法 15.1 灰色赋值器屏障

(a)Steele[1975, 1976] 屏障

```
1 atomic Write(src, i, ref):
2   src[i] ← ref
3   if isBlack(src)
4     if isWhite(ref)
5       revert(src)
```

(b)Boehm 等 [1991] 屏障

```
1 atomic Write(src, i, ref):
2   src[i] ← ref
3   if isBlack(src)
4     revert(src)
```

(c)Dijkstra 等 [1976, 1978] 屏障

```
1 atomic Write(src, i, ref):
2   src[i] ← ref
3   if isBlack(src)
4     shade(ref)
```

算法 15.2 黑色赋值器屏障

(a)Baker [1978] 屏障

```
1 atomic Read(src, i):
2   ref ← src[i]
3   if isGrey(src)
4     ref ← shade(ref)
5   return ref
```

(b)Appel 等 [1988] 屏障

```
1 atomic Read(src, i):
2   if isGrey(src)
3     scan(src)
4   return src[i]
```

(c)Abraham 和 Patel [1987] / Yuasa [1990] 屏障

```
1 atomic Write(src, i, ref):
2   if isGrey(src) || isWhite(src)
3     shade(src[i])
4   src[i] ← ref
```

Hellyer 等 [2010], doi: 10.1145/1806651.1806666.

© 2010 Association for Computing Machinery, Inc., 经许可后转载

15.2.1 灰色赋值器屏障技术

我们首先考虑灰色赋值器需要用到的屏障技术。此类解决方案均使用插入屏障 (insertion barrier)^①来拦截向黑色对象写入白色指针的操作，从而满足强三色不变式的要求。由于赋值器本身是灰色的，所以不需要额外的读屏障。此类解决方案均属于增量更新技术。

- Steele[1975, 1976] 提出了算法 15.1a 所示的屏障技术。该方案仅简单地关注写操作所修改的来源对象，因而其在所有方案中精度最高。该方案不会改变任何对象的可达性，但它却会将黑色来源对象重新着为灰色，进而实现追踪波面的后退。对于白色目标对象而言，其可达性的判定将被推迟到回收器重新扫描来源对象的过程中（在重新扫描来源对象之前，该指针可能又被赋值器删除）。由于该方案会导致回收波面的后退，所以其相当于是以牺牲回收进度为代价来换取回收精度。

① 我们认为，“插入屏障” (insertion barrier) 这一概念比“增量更新屏障” (incremental update barrier) 更加清晰。类似地，我们更加喜欢“删除屏障” (deletion barrier) 这一术语而非“起始快照屏障” (snapshot-at-the-beginning barrier)。

- Boehm 等 [1991] 实现了 Steele [1975, 1976] 屏障的一个变种, 如算法 15.1b 所示, 其不同之处在于该屏障会忽略新插入指针的颜色。该屏障最初的实现版本是通过虚拟内存脏标记来记录脏页, 从而无需显式地对赋值器在堆中的写操作进行拦截, 但这一版本的写屏障不会事先判断其所回退的对象是否为黑色, 因而其精度较低。在回收结束阶段, Boehm 等需要通过万物静止的方式来重新扫描脏页。
- Dijkstra 等 [1976, 1978] 提出了算法 15.1c 所示的屏障技术。对于插入到黑色对象中的白色指针, 不论其在未来是否会被赋值器删除, 该屏障都会将其标记为可达 (即着色)。与 Steele 的屏障技术相比, 尽管该策略的精度较低, 但它却有助于回收波面的推进。该屏障的最初版本在对目标对象进行着色时会忽略来源对象的颜色, 进一步降低了回收精度, 但忽略这一额外检测的好处在于可以省略对原子性的依赖, 因为写入指针和着色这两步操作各自天然都是原子化的。此处还有一个十分微妙的问题需要注意, 即指针写入操作必须先于着色操作执行。如果将这两步操作颠倒, 那么在 `ref` 被着色之后、写入 `src` 之前, 一旦回收器启动新一轮回收, 其便会将包括 `ref` 在内的所有对象重置为白色。一旦回收器扫描 `src` 的操作发生在屏障的写入操作之前, 则会出现从黑色对象指向白色对象的指针, 这显然违背了强三色不变式的要求 [Stenning, 1976]。

315

15.2.2 黑色赋值器屏障技术

接下来, 我们将介绍三种黑色赋值器相关的屏障技术。前两种屏障均使用增量更新策略来满足强三色不变式的要求, 它们都需要使用读屏障来阻止赋值器载入白色指针 (即拦截将白色指针插入到黑色赋值器的行为); 第三种屏障基于起始快照技术, 其使用删除屏障 (deletion barrier) 来拦截堆中的指针写操作, 并以此满足弱三色不变式的要求 (即阻止赋值器将起始快照中某一可达对象的唯一可达路径破坏)。弱三色不变式允许黑色赋值器持有白色指针, 赋值器被着为黑色意味着回收器无需对其进行重新扫描, 因此如果某一白色对象被黑色赋值器所引用, 其必然处于灰色保护状态。

- Baker [1978] 提出了算法 15.2a 所示的读屏障 (赋值器插入屏障)。对于回收过程中被赋值器读取的白色对象, 即使赋值器并未将其插入到追踪波面之后, 该屏障仍会将其着色, 因而其精度会比 Dijkstra 等的方案更低。该屏障最初是为复制式回收器而设计, 其着色操作会将白色对象从来源空间复制到目标空间, 因而 `shade` 方法所返回的将是目标空间中对象新副本的地址。
- Appel 等 [1988] 实现了 Baker 读屏障的一个粗粒度变种 (即精度更低), 如算法 15.2b 所示。针对赋值器访问堆中灰色页的操作, 该方案使用操作系统的虚拟内存页保护原语为其设置陷阱, 从而省去了软件层面的读屏障。陷阱处理函数会对灰色页进行扫描 (并解除保护), 然后再恢复赋值器的执行。该屏障可以应用在复制式回收器中, 因为其扫描操作可以对来源对象中所有指向来源空间的指针进行转发, 包括赋值器正在加载的域。
- Abraham 和 Patel [1987]、Yuasa [1990] 各自独立提出了算法 15.2c 所示的删除屏障。如果在图 15.2 所示的两种情况中使用该屏障, 则 D2 操作将直接把对象 `z` 着为灰色, T2 操作将把对象 `r` 着为灰色, 因而对象 `s` 最终可以得到着色。在回收过程中, 即使赋值器删除了指向某一对象的最后一个指针, 从而导致该对象不可达, 删除屏障依然会将其着色, 因而该屏障的精度最低。对于插入屏障所保留的对象, 回收器至少

可以确定赋值器曾在其中执行了某些回收相关操作（获取或写入对象的引用），但删除屏障所保留的对象却不一定被赋值器操作过。例如在图 15.2b 中，为对象 *r* 着色将导致其成为浮动垃圾（因为其不再可达但无法得到回收），而这一操作唯一的目的却是为了保留对象 *s*。在这一快照屏障的最初实现版本中，着色操作是无条件的，即无论来源对象是何种颜色，该屏障都会将覆盖指针的目标对象着色。Abraham 和 Patel 基于虚拟内存写时复制机制实现了这一快照屏障。

15.2.3 屏障技术的完整性

Pirinen [1998] 指出，上述各种屏障技术已经囊括了并发回收中所有可能的屏障策略，同时他还提出了将读写屏障相结合的方案，如算法 15.3 所示。该方案为黑色赋值器引入了读屏障，同时在堆中设置了删除屏障。该方案中，如果堆中存在从黑色对象指向白色对象的指针，则必然存在至少一个灰色对象会直接引用该白色对象，因而其满足弱三色不变式的要求（这一保障比弱三色不变式的基本要求要稍强一些，因为后者只要求从灰色对象到白色可达对象之间存在至少一条指针链，而非指针）。白色可达对象要么会在回收器扫描灰色对象的过程中得到着色，要么会在其灰色来源对象被修改时由写屏障着色，因而黑色赋值器可以安全地从灰色对象中加载白色指针。读屏障可以确保赋值器永远不会从白色对象中加载白色指针。因此在整个回收周期中，对于任何一个白色可达对象，都会存在至少一个灰色对象（在某一时刻）直接持有其引用。

算法 15.3 Pirinen [1998] 黑色赋值器混合屏障

```

1  atomic Read(src, i):
2      ref ← src[i]
3      if isWhite(src)
4          shade(ref)
5      return ref
6
7  atomic Write(src, i, ref):
8      if isGrey(src)
9          shade(src[i])
10     src[i] ← ref

```

对上述各种屏障技术中的某些步骤进行精简（short-circuite）或者粗化（coarsen）^①，还可以衍生出一些新的变种，包括：

- 在将某一对象着为灰色时，可以趁机完成该对象的扫描并将其着为黑色。
- 对于在写操作之前扫描（包含被删除指针的）来源对象并将其着为黑色的写屏障，可以使用将被删除指针着为灰色的删除屏障来替代，且后者粒度更粗。
- 对于在读操作之前扫描来源对象并将其着为黑色的读屏障（即 Appel 等的屏障），可以使用将目标对象着为灰色的读屏障来替代（即 Baker 等的屏障），且后者粒度更粗。
- 对于将来源对象回退到灰色的屏障（即 Steele 和 Boehm 等的屏障），可以使用将被插入指针的目标对象着为灰色的插入屏障来替代（即 Dijkstra 等的屏障），后者粒度更粗，精度更低。

对于所有基于强三色不变式（即增量更新技术）的并发回收策略，其要么必须确保灰色

① 所谓粗化，是指以牺牲屏障精度为代价来降低屏障的执行开销。——译者注

赋值器不会在黑色对象中插入白色指针，要么必须确保黑色赋值器不会获取或者使用白色指针。此二条件符合其一便可满足强三色不变式的要求。

我们知道，基于弱三色不变式（即起始快照技术）的并发回收策略必须与黑色赋值器配合使用。在弱三色不变式的框架下，回收器不能再将灰色对象简单看作是通往白色可达对象的中间路径，对于被黑色对象所引用的白色对象而言，灰色对象还是它们可达路径上必需的占位节点。因此，快照屏障必须保留所有被灰色对象直接引用的白色对象，为此，当赋值器从灰色对象中删除白色指针时，快照屏障至少应当将被删除指针的目标对象着色。

对于经由一条白色指针链从灰色对象可达的白色对象（该对象也可能直接被黑色对象引用），我们要么必须阻止赋值器获取该路径上的白色对象以避免赋值器破坏该路径 [Pirinen, 1998]，要么必须确保赋值器从白色对象中删除引用时至少应将被删除指针的目标对象着色 [Abraham and Patel, 1987; Yuasa, 1990]。

综上所述，我们所介绍的各种屏障技术均可以满足其所维护不变式的最小要求，同时对这些屏障技术进行精简或者粗化也可得到一些新的变种。

15.2.4 并发写屏障的实现机制

不论是基于强三色不变式还是弱三色不变式，写屏障都必须探测出所有将回收相关指针写入对象域的操作，并使用某种数据结构来记录该指针的来源、目标或者该操作所覆盖的引用。但是，当赋值器向该数据结构中添加引用时，并发执行的回收器可能会同时从中移除引用并进行追踪，因此面对赋值器之间、赋值器和回收器之间的竞争，系统必须确保指针插入和删除操作的效率以及正确性。

记录灰色对象的一种方式是将其添加到日志中，我们曾在第 13 章介绍了多种可以高效满足这一要求的并发数据结构，本节我们主要关注的是另一种常见技术：卡表。第 11 章已经介绍过万物静止回收器中基本的卡表操作，本节我们将讨论在并发回收场景下卡表操作的复杂性及其解决方案。

318

我们曾在第 11 章介绍过如何使用卡表来实现记忆集（remset），其基本策略是将卡表中的每个字节与一小段连续堆空间（例如 512 B）相关联。卡表既可以用于分代回收器，也可以用于并发回收器。对于需要记录到卡表中的灰色对象，写屏障会将其所处的卡在卡表中的对应字节打上脏标记。与此同时，回收器也会并发地扫描卡表，找出灰色对象并对其进行追踪，最后清除脏标记。这一过程中，赋值器和回收器之间的竞争显然会影响回收的正确性。

灰色实体[⊖]的产生方式取决于回收器以及写屏障的类型。在分代回收器中，一旦在年老代对象的某个域中写入年轻代对象的引用，则该域将被着为灰色。对于使用 Steele 式后退屏障的并发回收器，一旦在已标记（黑色）对象中插入未标记（白色）对象的引用，则该对象将从黑色回退到灰色。对于使用 Dijkstra 式前进屏障或者 Yuasa 式删除屏障的并发回收器，其必须将所有位于脏卡中的对象当作灰色对象，尽管这将导致存活对象附近的垃圾得到保留并降低回收精度，但实践中这通常不会成为问题，正如 Abuaiadh 等 [2004] 所发现的，在整理式回收中，与以对象为单位的整理策略相比，以小内存块为单位进行整理仅会产生很小的空间浪费（参见 14.8 节）。

卡表即为并发回收器的工作列表，回收器必须对其进行扫描，从中找出脏卡并进行清理，直到所有的卡都得到清理为止。由于赋值器可能会在回收器清理完某个卡之后再次将其打上脏

⊖ 包括对象和域。——译者注

标记, 所以回收器必须重复扫描卡表。一种替代方案是将卡表的处理推迟到一个最终的万物静止阶段中, 但这很可能导致追踪过程的并发处理时段过早地结束 [Barabash 等, 2003, 2005]。

15.2.5 单级卡表

单级卡表是最简单的卡表实现方式。每个卡可能处于以下 3 种状态中: 脏 (dirty)、处理中 (refining)、已清理 (clean)。赋值器写屏障可以使用简单的存储指令将某个卡标记为脏, 从而不必借助于 CompareAndSwap 等原子指令 [Detlefs 等, 2002a]。当回收线程发现脏卡时, 它会先将其状态设置为“处理中”, 然后从中寻找回收相关指针, 最后再确定该卡的新状态。回收器通常可以简单地将卡的新状态设置为脏, 但 Detlefs 等还会对卡进行“汇总” (summarise) (参见第 11 章)^①。在回收器尝试写入卡的最新状态之前, 其必须判断该卡的状态是否依然是“处理中”, 否则便意味着在回收器的查找过程中赋值器再次将该卡设置为脏。如果其状态依然为处理中, 则回收器必须尝试原子化地更新状态 (例如使用 CompareAndSwap 原语), 如若失败, 则意味着赋值器并发地将该卡设置为脏, 此时卡中可能会包含新的未处理灰色对象。面对这一情况, Detlefs 等会简单地将卡的状态保持为脏, 然后继续处理下一个脏卡, 但也有其他一些策略会重新清理该卡。

15.2.6 两级卡表

大部分卡通常处于已清理状态, 因而两级卡表可以提升回收器在卡表中查找脏卡的效率。在粒度更粗的第二级卡表中, 每个条目均对应 2^n 个粒度更细的卡。清理两级卡表的过程与清理单级卡表类似, 即当回收器发现一个二级脏卡时, 其会先将该二级卡的状态设置为“处理中”, 然后再进一步查找其所对应的一级卡。当完成所有一级卡的清理之后, 回收器会尝试将二级卡的状态原子化地设置为“已清理”。需要注意的是, 这一过程中存在一个微妙的并发问题: 在负责清理卡的回收线程看来, 写屏障标记脏卡的操作并非原子化的, 因而写屏障必须先将一级卡标记为脏, 然后再标记其所对应的二级卡, 回收器则必须以相反的顺序读取两级卡的状态。为确保正确的执行顺序, 可能需要付出额外的内存屏障开销。

15.2.7 减少回收工作量的相关策略

Barabash 等 [2003, 2005] 提出了一种减少回收器冗余工作的方案, 即尝试避免多次扫描同一对象。在该方案中, 回收器将把清理卡的工作推迟到出现新的追踪工作时。这一主体并发回收器使用 Steele 式回退插入屏障, 回收器必须对脏卡中所有的已标记对象进行扫描并追踪其未标记的子节点。减少冗余扫描工作的第一种策略是在扫描脏卡时不去追踪其中的对象, 而是将其标记为“在清理时再进行追踪”。在后续清理过程中, 如果回收器在某一时刻发现其所处理的卡重新变脏, 其必须重新对卡中的所有对象进行追踪, 尽管这可能导致某些对象被二次追踪 (在这些对象得到首次追踪之后, 其所在的长才变脏), 但却保证了尚未追踪过的对象只会受到一次追踪。Barabash 等发现这一策略可以提升回收器的性能并减少高速缓存不命中的次数。需要注意的是, 在弱一致性平台上, 尽管内存访问顺序的变化可能使该优化策略失效, 但该技术的安全性依然可以得到保证。

减少冗余工作的第二种策略是降低脏卡的数量。我们知道, 如果 Steele 式插入屏障将某个卡标记为脏, 必然是因为赋值器修改了该卡中回收器已经追踪过的对象; 如果被修改的对

^① 汇总的目的是为了加速后续堆遍历过程的处理速度。——译者注

象尚未得到追踪, 则只要该对象依旧可达, 其终究会被回收器追踪到, 因而写屏障无需将其所在的卡标记为脏。也就是说, 写屏障无需将白色对象着为灰色^①。

卡表的优势在于其标记速度很快, 且无需原子操作, 其性能主要取决于查找回收相关指针的开销。提升卡表性能的第一种策略是无条件地将卡标记为脏, 同时额外使用一张表来记录每个卡是否包含已被追踪过的对象。如果卡表以字节数组而非位数组的方式实现, 则回收器可以通过如下方式周期性地清理脏卡, 整个过程无需任何原子操作:

```
for each dirty card C
    if not isTraced(C)                $
        setClean(C)
    if isTraced(C)
        setDirty(C)
```

提升卡表性能的第二种策略基于如下观察结果: 在许多应用程序中, 大多数指针写操作均是针对年轻对象的, 且这些年轻对象通常位于本地分配缓冲区中, 因此可以借助对象头部的一个位来反映其是否位于某个活动的本地分配缓冲区中。如果某一对象的该位得到设置, 则回收器将推迟对其的追踪过程, 并将其添加到延迟列表中。当分配缓冲区溢出时(即分配慢速路径), 赋值器会将该缓冲区内所有的卡设置为已清理, 同时清空该缓冲区中所有对象的延迟追踪位^②。Barabash 等发现, 回收器极少触达活动的本地分配缓冲区中的对象, 因而该策略可以大幅提升回收效率。

在弱一致性平台上还有一些额外的细节需要注意。为确保正确的执行顺序, 最简单的策略是回收器在将某个卡标记为已追踪之后、追踪某一对象之前均执行内存屏障; 清理脏卡时, 在回收器检查该卡是否为脏之后、判断该卡是否已被追踪过之前, 也需执行内存屏障。需要注意的是, 两种情况下均只有回收器线程需要执行内存屏障。另一种替代方案是清理线程从头扫描卡表, 清理并(在链表或者另一张卡表中)记录所有尚未完成追踪的脏卡。扫描完成后, 清理线程会与回收器进行握手并请求并发回收器执行某一同步屏障。握手过程完成后, 清理线程会重新扫描所有已被回收器追踪过但又被赋值器重新设置为脏的卡。

[320]

15.3 需要考虑的问题

不论是增量回收器(赋值器与回收器交替执行)还是并发回收器(赋值器与回收器并行执行), 其主要目的都是最大限度地缩短赋值器可以感知到的回收停顿时间。增量/并发回收策略要么需要赋值器执行一部分回收工作, 要么需要赋值器与回收器进行一定程度的同步(可能需要等待回收器完成某些工作), 因而通常需要付出额外的总体执行时间(即牺牲赋值器吞吐量)。在理想情况下, 并发回收器也许能够与赋值器完全并行, 进而缩短总体执行时间, 但天下没有免费的午餐, 并发回收技术必然要求赋值器和回收器之间进行某种程度的通信与同步, 其具体表现形式通常是赋值器屏障。另外, 赋值器和回收器之间在处理器时间以及内存方面的竞争(包括回收器对高速缓存的干扰)同样也可能降低赋值器的执行速度。

但在另一方面, 增量/并发回收也可能提升某些应用程序的吞吐量。回收器在单个赋值器操作上(即读取或写入操作)引入额外开销, 目的是为了减少用户可以感知到的回收停顿, 而应用程序的用户也可能是另一个对延迟十分敏感的程序。Ossia 等 [2004] 以三层事务处理系统为例指出, 万物静止式回收可能导致事务超时并引发重试, 而少量开销的引入(即执行

① 因而也无需将白色被修改对象所在的卡标记为脏。——译者注

② 此时该缓冲区中的灰色对象依然从延迟列表可达。——译者注

写屏障)即可避免更多的额外工作(即事务的超时处理逻辑)。

在后续章节中,我们将介绍多种并发回收策略,每种策略都会给赋值器引入不同程度的额外开销。并发引用计数回收会给指针加载和存储操作带来很高的额外负担;并发标记-清扫回收器不会移动对象,它给赋值器带来的指针访问开销(与屏障技术不同)相对较低,但它可能会受到内存碎片问题的影响;在移动式并发回收中,为确保赋值器不受回收器移动对象的影响,两者之间可能需要引入额外的同步机制,或者需要回收器将这一行为通知给赋值器;复制式回收器会带来额外的空间开销,并增大内存压力;在所有的并发回收器中,不论是读屏障还是写屏障,两者都会不同程度地影响赋值器的吞吐量,影响的程度取决于读写操作的频率以及屏障操作的工作量。

并发标记-清扫回收器通常会使用写屏障来通知回收器对某一对象进行标记;并发复制、并发整理回收器通常会使用读屏障来确保赋值器不会访问到已经存在新副本的陈旧对象。回收器需要在屏障操作的频率和每次操作的工作量之间进行平衡。对于可能引发复制与扫描的写屏障,其执行开销显然高于仅执行简单复制操作的写屏障,而后者的执行开销显然又高于仅需改变来源指针的写屏障。类似地,将部分工作提前可能会减少后续过程中写屏障的工作量。另外,所有这些因素都还取决于回收相关工作的执行粒度,这一粒度可能是单个引用,也可能是对象,还可能是页。

浮动垃圾问题也是并发回收算法需要考虑的问题之一。容许浮动垃圾的存在可以加速并发回收周期的结束,但这会带来额外的内存压力。

[321]

赋值器(线程)是否需要在回收周期的开始或者结束阶段挂起(前者是为了确保回收器找到所有赋值器根,后者是为了进行结束检测),也会影响赋值器的吞吐量。判定回收周期结束的标准也会影响浮动垃圾的多少。

大多数并发回收器仅可以在停顿时间和空间开销方面提供十分松散的保障。实时应用程序在空间和时间方面存在硬上限要求,这意味着系统必须在赋值器与堆的交互操作方面提供定义明确的前进保障,同时在应用程序的内存分配规模方面必须提供定义明确的空间保障。

增量回收与并发回收特别适用于存活对象比例很高的系统。在这一场景下,即使充分利用所有处理器来进行万物静止式并行回收也会产生不可接受的停顿时间。增量回收与并发回收的一个缺陷在于,只有在回收周期结束之后,不可达对象所占用的空间才能得到回收,因而堆中必须预留足够多的空间余量,或者给回收器分配足够多的处理器资源(从而减少了赋值器的可用处理器资源),进而确保回收周期可以在赋值器耗尽内存之前结束。我们将在第 19 章介绍实时回收时再讨论回收器的调度问题,届时我们将看到,这一问题的处理相当棘手。

一种替代方案是使用混合分代/并发回收器,该方案依然使用传统的分代策略来管理年轻代(即次级回收依然是万物静止式的),而年老代则使用并发回收器进行管理。这一方案存在诸多优势:年轻代的回收周期通常足够短(数毫秒),从而不会给赋值器的执行带来太大干扰;弱分代假说告诉我们,大多数对象都在年轻时死亡,因而大多数内存都将在下一个回收周期中立即得到回收,从而减少了堆中预留的空间余量(其目的在于避免内存耗尽),降低了空间开销;由于年轻代使用万物静止式的回收策略进行管理,所以无需为其引入并发写屏障,仅在分代间引入写屏障便已经足够;尽管大多数新生对象并不会活到下一轮回收的开始,但并发回收算法通常会将新生对象着为黑色,这将导致其在下一轮回收中必然得到保留,而使用分代策略来分配新对象则轻易化解了这一问题;最后,年老代对象的修改频率会比年轻代低得多 [Blackburn and McKinley, 2003],这正是增量/并发回收理想的用武之地,

[322]

因为其写屏障的调用频率也会很低。

并发标记 – 清扫算法

第 15 章我们介绍了增量回收与并发回收的实现基础，同时也指出了此类回收器所遇到的共性问题。本章我们将关注此类回收器中的一个家族，即并发标记 – 清扫回收器。并发回收算法所面临的最重要的问题是回收的正确性，这要求赋值器和回收器之间必须进行适当的通信，从而确保它们能够在堆的拓扑结构方面保持一致的认知。赋值器有义务确保回收器不会将存活对象误判为不可达，而移动式回收器也有义务确保赋值器能够正确访问到已经移动的对象。

并发标记 – 清扫算法家族是最简单的一类并发回收器。由于回收器不会修改指针域，所以赋值器可以自由地从堆中加载指针，同时无需担心任何来自回收器的干扰，因此非移动式回收器天然不需要读屏障。赋值器在堆中的读操作通常要比写操作更加频繁，因此对于非移动式回收器而言，维护强三色不变式所必需的读屏障往往会引入很大的开销。例如，Zorn[1990]发现，SPUR Lisp 语言中指针读操作的静态频率为 13% ~ 15%，而指针写操作则为 4%，其测量结果表明，内联写屏障的运行时开销为 2% ~ 6%，而读屏障的开销则要达到 20%。这一普遍规律的唯一例外是依靠编译器优化技术来去除冗余屏障 [Hosking 等，1999；Zee and Rinard，2002]，并且将某些屏障操作合并到已有的空指针检查操作中 [Bacon 等，2003a]。基于上述原因，并发标记 – 清扫回收器通常使用 Dijkstra 等 [1976，1978] 的增量更新写屏障或者 Steele [1976] 的插入写屏障，也可使用 Boehm 等 [1991] 的粗化变种，或者 Yuasa[1990] 的起始快照删除屏障。

16.1 初始化

并发回收器通常不会等到赋值器耗尽内存时才开始执行，即使赋值器正在分配内存，并发回收器仍可以正常执行，但何时启动新一轮标记过程却比较难决定。如果回收周期开始的太晚，则可能会出现部分内存分配请求得不到满足的情况，此时赋值器只能挂起直到回收周期结束为止。一旦回收周期开始，则回收器的稳态工作率（steady-state work-rate）必须能够确保当前回收周期在赋值器耗尽内存之前结束，同时还应当尽量减少对赋值器吞吐量的影响。何时以及如何启动新一轮回收周期、如何确保并发回收过程中存在足够的可用分配内存、如何确保回收周期能够正常结束并将垃圾对象回收，这些问题的答案均取决于如何在赋值器工作的同时合理调度回收工作。

323

算法 16.1 展示了并发标记 – 清扫回收器中的赋值器分配过程，在该回收器中，（赋值器线程的）每个内存分配请求都会调用 `collectEnough` 方法来增量式地完成一部分回收工作。`atomic` 修饰词意味着这一工作需要与其他并发执行的赋值器线程或者回收器线程进行同步。`behind` 方法控制着何时执行回收工作以及每次执行多少，从而可以避免因赋值器比回收器超前过多而导致的内存分配失败。

算法 16.1 主体并发标记 – 清扫分配

```
1 New():  
2   collectEnough()
```

```

3   ref ← allocate()           /* 如果赋值器为黑色，则新分配的对象也必须着为黑色 */
4   if ref = null
5       error "Out of memory"
6   return ref
7
8   atomic collectEnough():
9       while behind()
10          if not markSome()
11              return

```

算法 16.2 展示了回收工作启动后回收器的执行过程。回收器首先清空工作列表，然后扫描赋值器根以建立最初的工作列表。假定回收器扫描赋值器根的过程需要将所有赋值器线程挂起，则当扫描完成后赋值器线程将不会引用任何白色对象。由于该算法需要一个万物静止式的回收初始化过程，因而其属于主体并发模式。初始扫描过程所遍历到的灰色根对象构成了后续扫描过程的初始回收波面。完成赋值器根的扫描之后，赋值器线程将恢复执行，其所使用的写屏障类型将决定赋值器究竟是黑色（即不允许赋值器持有任何白色引用）还是灰色。

算法 16.2 主体并发标记

```

1   shared worklist ← empty
2
3   markSome():
4       if isEmpty(worklist)                                /* 初始化回收过程 */
5           scan(Roots)                                     /* 不变式：赋值器不持有任何白色对象的引用 */
6       if isEmpty(worklist)                                /* 不变式：灰色对象已经全部处理完毕 */
7           /* 标记过程结束 */
8           sweep()                                         /* 立即清扫或懒惰清扫 */
9       return false                                       /* 标记结束 */
10      /* 回收过程继续 */
11      ref ← remove(worklist)
12      scan(ref)
13      return true                                         /* 后续过程需要继续进行标记 */
14
15   shade(ref):
16       if not isMarked(ref)
17           setMarked(ref)
18           add(worklist, ref)
19
20   scan(ref):
21       for each fld in Pointers(ref)
22           child ← *fld
23           if child ≠ null
24               shade(child)
25
26   revert(ref):
27       add(worklist, ref)
28
29   isWhite(ref):
30       return not isMarked(ref)
31
32   isGrey(ref):
33       return ref in worklist
34
35   isBlack(ref):
36       return isMarked(ref) && not isGrey(ref)

```


万物静止策略可能会引入不可接受的时间停顿。如果灰色赋值器屏障已经安装就绪，则可以简单地激活屏障并将扫描根的操作推迟到与赋值器并发执行阶段。我们将在 16.5 节介绍如何放宽将所有赋值器线程挂起并扫描其根这一要求，但尽管如此，回收器仍需对至少一个赋值器的根进行扫描，以便完成工作列表的初始化。

16.2 结束

如果只允许黑色赋值器的存在，则回收周期的结束将是一个相对直接的过程：当工作列表中不再包含待扫描灰色对象时，回收结束。此时，即使回收器遵从弱三色不变式，赋值器也只可能包含黑色引用，因为此时堆中将不存在任何从灰色对象可达的白色对象（因为已经不存在灰色对象）。赋值器为黑色意味着无需重新扫描其根。

如果允许灰色赋值器的存在，回收过程的结束则会稍加复杂，因为赋值器可能会在（初始化阶段的）根扫描完成之后重新载入白色指针，所以在回收结束之前，回收器必须重新扫描灰色赋值器的根。如果在重新扫描过程中没有发现任何新的灰色对象，则意味着回收结束。因此在算法 16.2 中，回收器在进入清扫阶段之前必须重新扫描赋值器根，以确保不存在更多灰色引用（第 5 行）。

16.3 分配

分配器必须依照赋值器的颜色来为新分配的对象赋予合适的标记状态（即颜色）。如果赋值器为黑色，则强三色不变式要求新分配的对象必须被着为黑色（即已标记）；而在弱三色不变式框架下，除非新分配对象从某个灰色对象可达，否则分配器也应将其着为黑色。由于分配器通常很难判定新分配的对象是否从灰色对象可达，所以即使是在弱三色不变式的框架下，分配器也通常会将新分配对象着为黑色 [Abraham and Patel, 1987 ; Yuasa, 1990]。相比之下，基于灰色赋值器的实现方案在分配对象时存在多种着色策略。

Kung 和 Song [1977] 发现，赋值器通常会很快将新分配对象链接到其他可达对象，此时他们所使用的无条件 Dijkstra 式增量更新写屏障便可在此时对其进行着色。基于这一原因，他们所设计的分配器仅在标记阶段分配黑色对象，在其他阶段则分配白色对象。另外，由于新分配对象不包含任何引用，因而分配器可以安全地将其直接着为黑色。

对于标记阶段的对象分配操作，Steele [1976] 会根据用于对象初始化的指针的颜色来确定新分配对象的颜色。假定分配器可以预知新分配对象各指针域的初始值，则其可以批量判断这些指针目标对象的颜色，如果它们均非白色，则分配器可以安全地将新分配对象着为黑色。另外，如果所有用于对象初始化的指针均非白色，很可能意味着标记过程已经接近尾声，同时也意味着新分配对象很可能在标记阶段结束之后依然存活。反之，如果所有的初始化指针均为白色，则回收器也会将新对象着为白色。Steele 的回收器将赋值器栈的扫描放在最后阶段，且其扫描方向为从栈底（数据变化最不频繁）到栈顶（数据变化最频繁），因此大多数新分配对象都将被着为白色，从而可以减少堆中的浮动垃圾。

在清扫过程中，Steele [1976] 的分配器会依照清扫器在堆中的遍历位置来决定新分配对象的颜色：从已经清扫过的区域中分配的对象将为白色，反之则为黑色（目的是避免清扫器将新分配对象误认为垃圾）。

将新分配对象着为白色而非黑色可能会引发一个问题，即这些白色对象久而久之可能会积累成一条很长的白色对象链，如果这些对象依然可达，则在当前回收周期结束之前回收器

终究需要对其进行追踪（例如当灰色赋值器分配了一个较大的白色数据结构时）。尽管将新分配对象着为黑色能够确保回收过程的结束不会因此受到延迟，但这一策略却会导致新分配的短命对象只能在下一轮回收周期中得到回收，从而造成空间上的浪费 [Boehm 等, 1991; Printezis and Detlefs, 2000]。Vechev 等 [2006] 提出了一种折中方案，该方案为新分配的对象赋予了新的颜色（第 4 种颜色）：黄色。清扫器可以将黄色对象等价于白色对象（它们可能会在当前回收周期结束之前就已经死亡），而标记器则会将其当做黑色对象，也就是说，标记过程可以直接把黄色对象着为黑色，同时无需对其进行扫描。因此，在回收结束阶段对灰色赋值器的追踪过程中，如果遇到黄色对象，则意味着无需进一步对其进行追踪。

16.4 标记过程与清扫过程的并发

目前为止，我们仅考虑了标记过程与赋值器之间的并发，同时假定标记过程与清扫过程之间是串行的。懒惰清扫意味着赋值器会在内存分配过程中执行并发清扫，其所清扫的内存块是由上一次标记过程所确定的，而在懒惰清扫的过程中，新一轮回收的标记过程可能已经开始。这一情况可能会潜在地导致对象的颜色出现混乱。此处的解决方案是将上一轮标记过程所识别出的真正白色垃圾对象（需要进行清扫）与下一轮回收过程中尚未追踪到的（未标记）白色对象进行区分。Lamport [1976] 引入紫色来对上述两种白色对象进行区分，从而将标记和清扫阶段的执行流水线化。标记阶段结束后，回收器会将所有白色垃圾对象重新着为紫色，而清扫过程将回收紫色对象并将其添加到对应的空闲链表中（同时需要重新将其着为黑色或者白色，具体取决于分配颜色）。

326

对于存在多个并发标记与清扫线程的场景，Lamport 的回收器在每个回收周期中将通过如下方式进行工作：

- 1) 等待所有的标记器与清扫器结束工作。

- 2) 将所有白色节点着为紫色，然后将所有黑色节点着为白色（白色有利于减少浮动垃圾）或灰色（此时该节点已经由赋值器写屏障进行并发着色）。

- 3) 将所有赋值器的根着色。

- 4) 启动标记器与清扫器。

标记过程将忽略所有紫色对象：赋值器永远不会获取紫色对象的引用，因而灰色对象永远不会引用紫色对象，且紫色对象永远不会被着色。这一策略的实现难点在于，在清扫过程开始之前，回收器必须遍历所有垃圾对象并将其着为紫色，类似地，在标记过程的开始阶段，回收器必须将（上一个回收周期所识别出的）所有黑色对象重新着为白色。

针对这一问题，Lamport 提出了一种优雅的解决方案，该方案会在回收过程的第 2 步为每个对象当前的颜色值赋予新的含义。每个对象将对应一个占用两个位的颜色标签 `hue`（白色、黑色、紫色），外加一个占用一位的 `shaded` 旗标。如果 `hue` 为白色，则设置 `shaded` 旗标位意味着将对象着为灰色（即：`shaded` 旗标被设置 + `hue` 为白色 = 灰色）；如果 `hue` 为黑色，则设置 `shaded` 旗标无任何作用（即只要 `hue` 为黑色，则不论 `shaded` 旗标是否已被设置，对象均为黑色）；如果 `hue` 为紫色，则由于回收器永远不会追踪到垃圾对象，因而其 `shaded` 旗标不可能被设置。`hue` 值的具体含义是由全局变量 `base` 所决定的，`hue=base` 意味着白色，`hue=base+1` 意味着黑色，`hue=base+2` 意味着紫色。在回收的第 2 阶段，由于标记过程与清扫过程均已经结束，所以堆中不存在任何灰色或者紫色节点，此时如果要将黑色对象翻转为白色，以及将白色对象翻转为紫色，只需简单地增加变量 `base` 并将其模 3。表 16.1 展示了变量

base 可能存在的 3 个值 (00, 01, 10) 以及 shaded 旗标可能存在的两个值 (0, 1), 此二变量组合起来便可表示所有对象的颜色。value 列中的每个值模 3 之后即为对象真实的 hue 值。需要注意的是, 由于回收器永远不会将紫色 (垃圾) 对象着色, 所以不可能存在 hue=base+2 且 shaded=1 这一情况。后续增量回收周期中 hue 值的含义将依照相同的方式解析。

表 16.1 Lamport[1976] 的颜色标记方案

tag		GC1: base=00		GC2: base=01		GC3: base=10	
hue	shaded	value	colour	value	colour	value	colour
00	0	base	白色	base+2	紫色	base+1	黑色
00	1	base	灰色	base+2	N/A	base+1	黑色
01	0	base+1	黑色	base	白色	base+2	紫色
01	1	base+1	黑色	base	灰色	base+2	N/A
10	0	base+2	紫色	base+1	黑色	base	白色
10	1	base+2	N/A	base+1	黑色	base	灰色

注: 该方案在标记过程和清扫过程并发的场景下使用“hue 和 shaded”的颜色编码方案。hue=base 且 shaded=0 代表白色, hue=base 且 shaded=1 代表灰色, hue=base+1 代表黑色, hue=base+2 代表紫色。(紫色) 垃圾对象的 shaded 旗标永远不会被设置。当所有标记器与清扫器都执行完毕时, 堆中将不存在灰色或紫色对象, 因而只需简单地增加 base 变量并将其模 3, 便可实现从黑色到白色 / 灰色、从白色到紫色的颜色翻转。

327

为确保在阶段 2 结束时堆中不会有灰色对象存留到下一轮回收 (除非该节点是由赋值器刚刚着为灰色), 标记线程在将灰色节点着为黑色的同时也应当清除其 shaded 旗标, 否则这些灰色对象将在后续回收过程中成为浮动垃圾。另外, 为加速垃圾对象的识别, 标记器与清扫器在遇到黑色对象时也可清除其灰色旗标。

Queinnec 等 [1989] 提出了一种替代方案, 即在奇数轮和偶数轮回回收周期中使用相互独立的颜色信息, 因此当前回收周期的标记过程可以与上一轮回回收周期的清扫过程相互独立地执行。

16.5 即时标记

到目前为止, 我们均假定在标记阶段的初始化或者结束过程中, 回收器都需要将所有赋值器线程挂起以扫描其根。根的初始扫描结束之后, 赋值器将不会引用任何白色对象, 此时赋值器既可以保持黑色状态继续运行 (前提是使用黑色赋值器屏障), 也可在灰色状态下 (前提是使用灰色赋值器屏障) 继续运行 (但为确保标记过程的正确结束, 回收器最终必须挂起灰色赋值器并重新扫描其根, 直到其不再引用任何灰色对象为止)。这些万物静止式的操作降低了回收的并发程度。另一种策略是独立地对每个赋值器的根进行采样, 此过程可以与其他赋值器线程并发执行, 但这一策略会引入额外的复杂度: 同一时刻某些赋值器线程将在灰色状态下执行, 而其他赋值器线程则将在黑色状态下执行, 且算法的结束判定可能会因此受到影响。

即时回收永远不会同时挂起所有赋值器线程, 相反, 回收器和赋值器之间可以通过一系列软弱手 (soft handshake) 机制来实现后者根的扫描。这一过程并不需要回收器发出全局性的硬同步指令, 其只需要逐个地、异步地驱使赋值器线程在某个合适的位置优雅地挂起, 然后再扫描 (甚至修改) 其当前状态 (栈与寄存器), 最后再恢复其执行。在一个赋值器线程被

挂起的同时，其他赋值器线程可以不受干扰地继续执行。另外，如果进一步使用第 11.5 节所介绍的栈屏障技术，则当线程被挂起时，回收器可以仅对其栈顶的活动记录栈帧进行检查（所有其他栈帧可以借助于栈屏障技术进行异步扫描），因而其握手速度会非常快，最大限度地减少了停顿时间。

16.5.1 即时回收的写屏障

即时回收器中同步操作的设计应当十分谨慎。对于挂起所有赋值器线程并扫描其栈的主体并发回收器而言，一种常见的实现方案是使用黑色赋值器并辅以删除屏障，同时将新分配的对象着为黑色。在该方案中，回收器无需重新扫描黑色对象，且分配器不会给回收器带来更多工作，因而简化了标记阶段的结束过程。但是，该方案却不能满足即时回收器的要求，如图 16.1 所示。由于扫描过程是即时的而非万物静止式的，所以有可能出现黑色赋值器与白色赋值器共存的情况。将新分配对象着为黑色，意味着在回收器完成所有赋值器线程的扫描之前，也就是追踪过程开始之前，堆中便已经可能存在黑色对象。由于栈操作无法触发删除屏障，且不存在插入屏障，所以对象 X 和 Y 都无法被着为灰色。总之，为即时标记过程设计正确的赋值器 - 回收器同步机制是一个十分复杂的问题，它需要算法的设计者必须十分小心地避免错误。

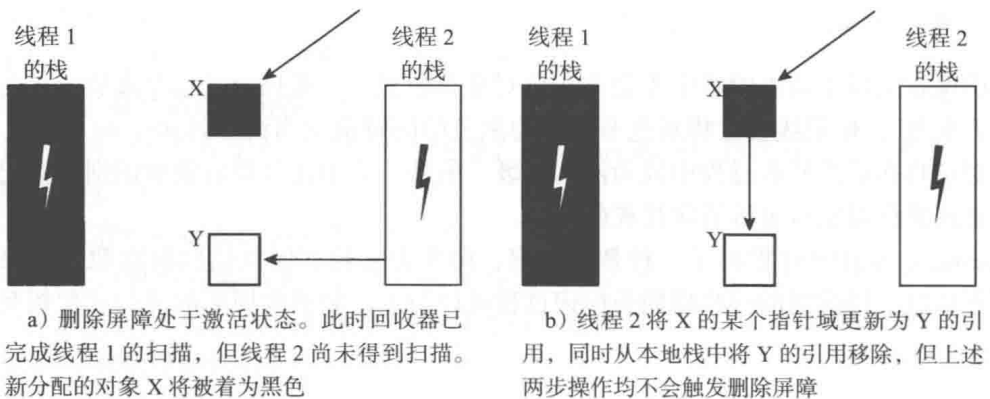


图 16.1 对于即时回收器而言，如果仅使用“将新分配对象着为黑色，且使用删除屏障”这一策略，堆中仍有可能存在某一白色对象仅从黑色对象可达

328

16.5.2 Doligez-Leroy-Gonthier 回收器

使用软握手机制来初始化标记过程的方法最早应用在针对 ML 编程语言的标记 - 清扫回收器中，根据其作者的名字 [Doligez and Leroy, 1993 ; Doligez and Gonthier, 1994]，该回收器被命名为 Doligez-Leroy-Gonthier 回收器。该回收器使用线程本地堆，因此回收器可以对仅被单个线程使用而不与其他线程共享的数据进行独立回收。系统使用一个全局堆来存储线程共享对象，同时要求全局堆中的对象不得包含任何指向线程私有堆的指针。一旦某一线程本地堆中的对象被其他线程引用，系统的动态逃逸检测机制便会捕获这一信息并将其复制到共享堆中。回收器只允许不可变对象在线程本地堆中分配（ML 语言中绝大多数对象都是不可变对象），因此系统在将它们复制到全局共享堆时无需更新其所有来源引用（但仍需复制可达对象的传递闭包）。但由于 ML 中的对象修改操作很少，因而逃逸现象也较少发生。这些规则允许回收器在仅挂起一个赋值器线程的情况下独立完成其堆的回收。

为避免更新每个线程指向共享堆的引用，Doligez-Leroy-Gonthier 使用并发标记 - 清扫策略来管理共享堆。当并发标记 - 清扫回收器稳定工作时，Yuasa 式快照删除屏障将保证赋值器始终处于黑色状态。在回收工作达到稳定状态之前，回收器需要通过一系列软握手机制来将赋值器线程从灰色转变为黑色，其具体流程如下。

回收器以及每个赋值器线程都通过一个私有状态变量来反映其所感知到的回收状态。为启动新一轮回收周期，回收器将自身状态设置为 Sync₁，其他赋值器线程可以通过软握手机制感知到这一信息，并据此更新自身状态。一旦所有赋值器线程均完成对 Sync₁ 软握手请求的应答，则回收器将真正处于 Sync₁ 阶段。赋值器线程在设置某一指针域或者分配过程中将忽略软握手请求，目的是为了确保在进行软握手之前这些操作已经执行完毕，进而确保这些操作在状态变更时的原子性。Sync₁ 软握手完成后，每个赋值器线程将执行算法 16.3a 所示的写屏障，该屏障会对被修改指针域的新旧目标对象分别进行着色，其本身相当于是黑色赋值器 Yuasa 式起始快照删除屏障和灰色赋值器 Dijkstra 式增量更新插入屏障的混合体。赋值器并不会将其所着色的对象直接添加到回收器的工作列表中以进行扫描（Kung and Song[1977] 便采用这一策略），而只是简单地将白色对象着为灰色，然后再设置全局 dirty 旗标来通知回收器扫描新的灰色对象（类似于 Dijkstra 等 [1978] 所使用的策略）。尽管这一策略可以避免赋值器和回收器之间进行显式同步（与软握手机制不同，软握手机制实现原子化的方法是简单地推迟对握手请求的应答），但其同时也意味着在最坏情况下，标记阶段的结束过程需要重新扫描整个堆以找出其中的灰色对象，由于 ML 中的对象修改操作极少，所以这一问题不会对算法的实现造成太大影响。Sync₁ 软握手完成后，灰色赋值器线程所分配的对象依然为白色（与回收周期开始前的分配颜色相同）。

329

算法 16.3 Doligez-Leroy-Gonthier 写屏障（此处均忽略了握手操作）

a) 同步屏障	b) 异步屏障
<pre>1 WriteSync(src, i, new): 2 old ← src[i] 3 shade(old) 4 shade(new) 5 src[i] ← new</pre>	<pre>1 WriteAsync(src, i, new): 2 old ← src[i] 3 if not isBlack(old) 4 shade(old) 5 if old ≤ scanned 6 dirty ← true 7 src[i] ← new</pre>

当所有赋值器线程都完成对 Sync₁ 握手的应答之后，回收器将进入下一个阶段，即 Sync₂ 阶段，该阶段同样需要经过一轮握手过程。设置写屏障操作的原子性只是针对握手过程的，这并不能保证所有赋值器线程同步完成写屏障的设置，这便可能导致如下一种情况的出现：当某个已经激活写屏障的线程执行完 old ← src[i] 操作之后[⊖]，其他尚未激活写屏障的赋值器线程[⊖]可能会立即将另一个指针 X 写入 src[i]，此时 shade(old) 所着色的对象将不是真正会被 src[i] ← new 这一操作所覆盖的指针 X。Sync₂ 阶段的引入正是为了解决这一问题，在进入该阶段之后，回收器便可以确定所有赋值器在 Sync₁ 握手之前的 Async 阶段已完成未被监控到的原子化分配操作或写操作，此时所有赋值器都已经在执行（带插入保护的）写屏障，因而即使写屏障的执行发生交错也不会产生任何问题。此时回收器将发起 Async 握

⊖ 即算法 16.3a 第 2 行执行完毕。——译者注
⊖ 即尚未完成 Sync₁ 握手。——译者注

手，每个赋值器响应 Async 握手的方法是：为回收器扫描自身的根（将自身着为黑色）、启用黑色对象分配、将写屏障切换为标准的快照屏障，该屏障会通过设置全局 dirty 旗标的方式（类似于 Dijkstra 等 [1978] 的方案）来通知回收器对可能插入到追踪波面之后的灰色对象进行扫描，如算法 16.3b 所示。此时回收器便可安全地进入稳态快照标记阶段，即 Async 阶段。

标记阶段结束后便可开始清扫过程。与 Steele[1975] 的策略类似，为减少浮动垃圾，清扫指针的位置将决定赋值器新分配对象的颜色：诞生于已清扫空间的新对象将为白色（该空间中得垃圾已经得到释放），反之则为黑色（以避免错误地将其清扫），如果新对象位于回收器正在进行并发清扫的区域，则其将为灰色（以避免与清扫器在两个区域的边界产生竞争）（见表 16.2）。

表 16.2 Doligez-Leroy-Gonthier 回收器中的状态变迁

阶段	回收器	赋值器	含义
Async	Async	Async	所有赋值器均未激活 Sync 屏障
	Sync ₁	Async, Sync ₁	某些赋值器可能在执行 Sync 屏障
Sync ₁	Sync ₂	Sync ₁ , Sync ₂	某些赋值器可能已经成为黑色，并且正在执行 Async 屏障；但其他赋值器仍在执行 Sync 屏障
Sync ₂	Sync ₂	Sync ₂	所有赋值器均在执行 Async 屏障
	Async	Sync ₂ , Async	某些赋值器为黑色，并且所有赋值器均在执行 Async 屏障
Async	Async	Async	所有赋值器均为黑色，回收器可以开始进行标记、扫描、清扫等操作

16.5.3 Doligez-Leroy-Gonthier 回收器在 Java 中的应用

Domani 等 [2000] 讨论了如何在 Java 环境下使用 Doligez-Leroy-Gonthier 回收器，他们需要额外考虑 Java 环境下较高的对象变更率，以及诸如弱引用和终结机制等语言相关特征。由于 Java 通常不支持不可变对象，所以他们不再使用独立的线程本地堆回收策略，而是简单地330对全局共享堆进行即时回收。原版 Doligez-Leroy-Gonthier 回收器假定处理器满足顺序一致性模型，而 Domani 等人的改进则可以在内存一致性更弱的多处理器上正确执行。为避免回收器重新扫描刚刚被赋值器着为灰色的对象（此类对象在诸如 Java 等以对象变更为导向的语言中十分普遍），Domani 等为每个赋值器线程配备了一个输出受限双端队列（output-restricted double-ended queue），赋值器可以将灰色对象压入队列的一端，回收器则可以使用轮询的方式从另一端获取工作，这一策略可以最大限度地减少写屏障中赋值器与回收器之间的同步开销。

16.5.4 滑动视图

Azatchi 等 [2003] 在即时标记方面做了进一步探索，他们使用滑动视图（sliding views）策略来对赋值器根进行采样，该策略无需引入万物静止式的停顿 [Levanoni and Petrank, 1999]。相对于 Domani 等 [2000] 所使用的双端队列，滑动视图策略实现快照删除屏障的方法是将标记过程中某一对象被修改（着色）之前所有域的状态记录到线程本地缓冲区中，缓冲区中的元素将通过软握手的方式一次性全部转移给回收器。当所有缓冲区均为空时，标记阶段结束。与 Doligez-Leroy-Gonthier 回收器类似，在首次握手之后、所有赋值器的删除屏障都得到激活之前，赋值器还需要执行一个 Dijkstra 式增量更新插入屏障，目的是避免在赋值器快照获取完成之前出现无法被写屏障感知到的指针写操作。被“窥探”写操作（snooped store）所写入的引用也将成为新的追踪来源[⊖]。在回收器确定所有赋值器线程都已经开始记录快照之后，“窥探”写入即可停止。进一步细节我们将在 18.5 节讨论。

⊖ 所谓“窥探”是指写操作在覆盖目标域之前先读取其原有的指针，如算法 16.3a。——译者注

16.6 抽象并发回收框架

不同的并发回收器之间存在许多相同的设计特征与机制，它们只是在一些细微但重要的细节上存在差异，因此我们可以采用一种并发垃圾回收通用抽象框架 [Vechev 等, 2005, 2006; Vechev, 2007] 来凸显它们之间的相同点以及不同点。我们知道，并发回收器的正确性取决于赋值器和回收器在并发状态下的协作方式，因此抽象并发回收器将所有与回收器和赋值器相关的事件记录到共享链表 `log` 中，这些事件包括：

- **T**`<src, fld, old, new>`：回收器已经追踪 (Trace) 过来源对象 `src` 中的指针域 `fld`，该域原本记录的是对象 `old` 的引用，而回收器将其修改为新的引用 `new`，也就是说，回收器扫描对象图中的边 `src → old`，并将其替换为 `src → new`。
- **N**`<ref>`：赋值器分配了新 (New) 对象 `ref`。
- **R**`<src, fld, old>`：赋值器在堆中发起一次读 (Read) 操作，该操作从来源对象 `src` 的 `fld` 域中加载了引用 `old`。
- **W**`<src, fld, old, new>`：赋值器在堆中发起一次写 (Write) 操作，该操作将引用 `new` 写入来源对象 `src` 的 `fld` 域中，该域中原本记录的值是 `old`。如果 `fld` 是一个指针域，则该操作会将对象图中的边 `src → old` 替换为新的边 `src → new`。

[331]

此处的 `src`、`fld`、`old`、`new` 分别为来源对象的地址、来源域的地址、老/新目标对象的地址。回收器事件集合 **T** 所记录的是已被赋值器扫描过的域集合，对于非移动式回收器而言，追踪过程不会修改堆中对象的引用，因而在事件集合 **T** 中 `old=new`。赋值器事件集合 **N** 记录的是赋值器的分配操作；赋值器时间集合 **R**、**W** 记录的是赋值器访问或修改过的域。

算法 16.4 描述了并发标记-清扫回收器的抽象算法，该算法脱胎于算法 6.1 所描述的抽象追踪式回收器，并针对回收器和赋值器并发执行的场景进行了改造。算法的执行过程依然符合传统的标记-清扫回收，即首先从根开始进行追踪，以便扫描所有可达对象，然后再通过清扫的方式回收不可达对象。扫描工作单元将在 `scanTracingInc` 方法中原子化地执行；执行清扫过程的 `sweepTracing` 方法也需要引入适当的同步机制，我们此处略去其具体实现细节。

在回收初始化阶段，回收器会通过 `rootsTracing` 原子化地对赋值器根进行采样，同时清空所有日志。为避免采样过程中赋值器线程并发地修改其根，该阶段使用原子化（即万物静止）的扫描方式以降低复杂度。而即时回收器则可以在不引入万物静止式停顿的前提下完成赋值器根的采样。

完成赋值器根的采样之后，赋值器将恢复执行并与回收器并发运行。回收器将不断地扫描对象，并且对赋值器并发写操作所产生的新的工作任务进行处理。

当扫描循环执行到一定程度时，回收器可以根据某些（此处暂时无法确定的）因素（记为②）来结束该循环。当回收器进入到结束阶段时，为确保当前回收周期的顺利结束，回收器必须原子化地完成剩余的回收工作（即阻止赋值器执行写操作）。扫描工作会在阻止所有并发写操作的情况下原子化地执行，这对于确保回收周期的顺利结束通常是必要的。但在实践中，某些算法可能并不需要这一原子化的结束阶段。

`scanTracingInc` 方法使用与传统回收器相同的方式来遍历堆，但其执行过程却是增量式的、与赋值器交替进行的。与算法 6.1 中的 `scanTracing` 方法的唯一不同之处在于，`scanTracingInc` 方法将回收器所遍历到的域及其所包含的引用添加到日志中的操作必须是原子化的。

`addOrigins` 方法调用了一个尚未定义的函数 `expose`，该函数可以根据日志中的首个元

素返回一个对象集合，回收器以该对象集合作为追踪源头 (origin)，从而会发现更多的存活对象。抽象并发回收框架是通过这一函数实现参数化的，即该函数的不同实现将会衍生出不同的抽象并发回收算法，每种不同的抽象算法可以对应文献中的一种具体回收算法，稍后我们将给出具体的实现案例。正是由于日志的存在，回收器才能避免扫描过程遗漏可达对象的情况，否则插入到回收波面之后的对象将无法得到标记。

算法 16.4 主体并发增量追踪式垃圾回收

```

1  shared log ← ()
2
3  collectTracingInc():
4      atomic
5          rootsTracing(W)
6          log ← ()
7      repeat
8          scanTracingInc(W)
9          addOrigins()
10     until ②
11     atomic
12         addOrigins()
13         scanTracingInc(W)
14         sweepTracing()
15
16  scanTracingInc(W):
17      while not isEmpty(W)
18          src ← remove(W)
19          if  $\rho(\text{src}) = 0$                                 /* 引用计数为零 */
20              for each fld in Pointers(src)
21                  atomic
22                      ref ← *fld
23                      log ← log · T(src, fld, ref, ref)
24                      if ref ≠ null
25                          W ← W + [ref]
26                       $\rho(\text{src}) \leftarrow \rho(\text{src}) + 1$         /* 增加引用计数 */
27
28  addOrigins():
29      atomic
30          origins ← expose(log)
31          for each src in origins
32              W ← W + [src]
33
34  New():
35      ref ← allocate()
36      atomic
37           $\rho(\text{ref}) \leftarrow 0$ 
38          log ← log · N(ref)
39      return ref
40
41  atomic Write(src, i, new):
42      if src ≠ roots
43          old ← src[i]
44          log ← log · W(src, &src[i], old, new)
45          src[i] ← new

```

16.6.1 回收波面

在并发场景下，回收器和赋值器之间需要密切协作以确保回收的正确性。日志记录了回

收器在堆中的追踪进度，其具体表现形式便是事件集合 T ，该集合同时也构成了回收过程的波面。协作的关键在于如何处理交替发生的赋值器事件（集合 N 、 R 、 W ），具体取决于这些事件所操作的堆中数据是已被回收器扫描过（即位于回收波面之后），还是尚未得到扫描（即位于回收波面之前）。回收波面本身是由待扫描域（而非这些域中所记录的指针的值）的集合所组成的。实践中的回收器通常会使用近似的方式来实现该波面，每种近似方式的精度都各不相同，例如以域为单位，或者以对象为单位、以页为单位、以其他物理或逻辑单元为单位等。

16.6.2 增加追踪源头

`addOrigins` 方法通过对日志的处理来发现新的存活对象，仅靠回收器的追踪过程有可能永远无法发现某些存活对象，因为赋值器操作可能会将其隐藏到回收波面之后。函数 `expose` 的具体实现决定了其所返回的追踪源头的精度。

16.6.3 赋值器屏障

`New` 和 `Write` 方法展示了赋值器所需执行的屏障（它们需要适当地进行原子化），在抽象并发回收框架中，它们的任务是将自身事件添加到日志中，并以此完成与回收器之间的协作。新分配对象将被记录到集合 N ，因而赋值器在后续执行过程中便可分辨出读 / 写某一新对象域的操作，也可分辨出以新对象的引用作为参数的读 / 写操作。新分配的对象通常仅具有唯一的引用来源（即来自赋值器的根——译者注），直到赋值器将其写入堆中的某个域为止。此外，新分配对象也不会包含任何引用（所有域都将被初始化为 `null`，直到其某个指针域得到修改为止）。具体的回收器可以根据这些特征来决定在回收周期中新分配对象的存活性：某些回收器会忽略这些对象的可达性，并将其全部当作存活对象，这将导致其中的不可达对象只能在下一轮回收中得到释放；其他回收器则只有当新分配对象的引用真正被写入存活对象之后，才会将其保留。

与非并发场景的写操作类似，`Write` 操作同样会执行 $\text{src}[i] \leftarrow \text{new}$ （且 $\text{new} \neq \text{null}$ ）这一操作，因而指向对象 `new` 的指针将被插入到对象 `src` 的 `src[i]` 域中，相应地，`src[i]` 中原有的指针 `old` 将从对象 `src` 中删除。如果 `src[i]` 位于回收波面之后，则指针 `new/old` 的插入 / 删除操作都将发生在回收波面之后。写操作集合 W 将同时捕获赋值器所插入以及删除的指针。

类似地，回收波面也可通过三色抽象来表示：位于回收波面之前的对象 / 域为白色；位于其上的为灰色，位于其后的为黑色。

16.6.4 精度

算法 16.4 所示的抽象并发回收器使用了固定的原子化级别（即带 `atomic` 前缀的代码段），`expose` 函数的不同实现将决定算法的精度。在抽象并发回收框架中对这一参数进行调整，我们即可获得各种文献中所描述的并发回收器的一个典型子集，但由于实践中的某些回收器在原子化方面与算法 16.4 并不完全一致，所以还有一些并发回收器无法直接由抽象并发回收的框架推导而来。例如，算法 16.4 假定回收器可以原子化地获取各赋值器线程的根，这便要求回收器必须使用同步的方式进行采样，其实现方式很可能是简单地将所有赋值器线程挂起（也就是说，算法 16.4 只是主体并发的）。

16.6.5 抽象并发回收器的实例化

如果要将抽象并发回收框架实例化,则需定义具体的 *expose* 函数。例如,Steele 式并发回收器需要重新扫描所有被修改对象并将其加入回收波面,从对象和域的级别来看,回收波面是通过对日志中每个条目所记录的对象/域进行追踪 (Trace) 而建立的;日志中每个写 (Write) 操作的 *src* 参数构成了被修改对象的集合, *fld* 参数则构成了被修改域的集合。在 Steele 式回收器中, *expose* 函数的实现是原子化地重新扫描已经追踪过但又被修改的域。传统的追踪方式是为每个对象设置一个标记位,此时回收波面的构成单元将是对象 (即追踪集合 *T* 中的 *src* 参数),但抽象并发回收框架同样允许以域为单位来构建回收波面,即允许通过某种机制对单个域进行标记,此时对于已经追踪过的对象,回收器便可仅对被修改的域而非整个对象进行重新扫描。另外,Steele 认为赋值器线程栈的数据变更十分频繁,因而对线程栈的重新扫描应当推迟到追踪过程的末尾。追踪过程的结束条件是,对于日志中的每条追踪记录 *T*,在位于该记录之后的其他所有记录中,不存在与 *T* 相对应的写操作 (Write) 记录 (以对象或者域为单位)[⊖]。

经典的 Dijkstra 式回收器会无条件地将所有写入堆中的引用着色,其 *expose* 函数的实现是:获取写操作记录中的参数 *new* 并将其添加到回收波面的追踪 (Trace) 记录中。需要注意的是,回收器可以直接从日志中获取 *new* 引用,同时无需重新扫描 *src/fld*。其追踪过程的结束条件与 Steele[1976] 类似。

对于 Yuasa 式快照回收器,其 *expose* 函数的实现是:对于日志中的每条写操作记录,如果在该记录之后不存在与之对应的追踪记录[⊖],则返回该记录中的 *old* 引用。该屏障将在赋值器修改目标对象之前创建追踪记录并完成追踪,因而与基于增量更新策略的屏障相比,其回收结束的速度更快。

16.7 需要考虑的问题

并发标记-清扫回收中的许多问题都属于所有并发回收器的共性问题。毋庸置疑,与万物静止式的回收器相比,并发回收器在设计、实现以及调试方面都更加复杂。因此在设计并发回收器时我们首先应当考虑,实际上是否需要引入这种额外的复杂度?是否应用诸如分代回收器等简单的解决方案就已经足够?

对于绝大多数的应用程序而言,分代回收器已经可以满足数毫秒级别的停顿时间要求,但在最差情况下 (即整堆回收时),其回收停顿时间可能会陡增,具体取决于堆空间大小以及存活对象总量等因素。这一较长的停顿时间可能无法令用户接受,相比之下,并发回收器的停顿时间更短、更具有可预测性。我们将在第 19 章看到,设计良好的实时回收器 (real-time collection) 可以确保其停顿时间小于毫秒级别,但鱼和熊掌不可兼得,此时赋值器和回收器通常都需要承担显著的额外开销。为了限制回收停顿时间,回收器不仅应当是并发的,还应当是即时 (on-the-fly) 的,即回收器仅应当在一次停顿中挂起一个赋值器线程并扫描其根。

并发标记-清扫回收所存在的其他问题与万物静止式标记-清扫回收类似,即所有非移动式内存管理器都会受到内存碎片问题的影响。复制式与整理式回收器除了能够解决碎片问

335

⊖ 更通俗地讲,即每个已经完成追踪的对象/域没有再被修改过。——译者注

⊖ 即写操作中的 *src/fld* 尚未得到扫描。——译者注

题，其所使用的顺序分配策略通常也会比空闲链表分配速度更快，且能够给赋值器提供更好的局部性。但标记-清扫回收器不需要额外的复制缓冲区，因而其空间利用会比复制式回收器要高。与其他并发回收算法相比，非移动式并发标记-清扫算法还存在一个显著优势，即它的堆一致性模型（heap coherency model）更加简单。不论是对于哪种并发回收器，为避免赋值器操作导致某一可达对象相对回收器不可见，赋值器均需将自身堆拓扑结构变化的信息通知给回收器。另外，移动式回收器不仅要确保同一对象只会被一个线程迁移，还应当确保在赋值器看来，更新被迁移对象全部引用来源的操作应当是原子化的。

并发标记-清扫回收器还允许设计者选择多种不同的具体实现策略。与其他并发回收器类似，新分配对象的颜色可以为黑色、灰色或者白色。黑色赋值器要求所有新分配对象的颜色均为黑色；灰色赋值器既可以使用白色分配，也可以使用灰色或者黑色分配，具体的着色策略取决于当前所处的回收阶段、新对象域的初始值、清扫器的处理进度等因素。

我们将在后续章节中探讨并发复制与并发整理回收器，最后将介绍可以满足硬实时系统停顿要求的回收器，其回收停顿时间能够满足系统所有的响应时限要求。

并发复制、并发整理算法

本章我们将介绍并发复制与并发整理回收算法，它们均可在赋值器执行的同时并发地移动对象，从而完成堆中碎片的整理。我们曾在第 3 章和第 4 章分别介绍过整理式与复制式回收，本章我们将考虑如何将其扩展到与赋值器并发执行的环境中。

我们首先关注基于复制的回收技术，即先将来源空间中的可达对象复制到目标空间，然后再将来源空间整体回收。回顾第 4 章我们可知，在扫描存活对象的过程中，回收器必须将所有来源空间指针重定向到目标空间，即：使用转发地址替换每个来源空间指针，并在首次发现来源空间中的某一对象时将其复制到目标空间。

并发复制回收不仅要确保回收器免受赋值器修改操作的影响，而且还要确保赋值器免受回收器并发复制对象的影响。另外，对于回收器正在复制的对象，回收器对其原始副本的修改必须能够同步到目标空间中正在创建的副本中。

对于复制式回收器而言，黑色赋值器意味着其只能持有指向目标空间的指针，一旦黑色赋值器持有了来源空间中的引用，由于它们永远不会被回收器重新扫描到，所以回收过程将必然出现错误。我们将这一结论称为黑色赋值器目标空间不变式 (tospace invariant)，即赋值器永远都只能操作目标空间中位于回收波面之前的对象。同理，从定义上讲，灰色赋值器在回收周期的初始阶段将仅持有来源空间指针，同时由于其不会使用读屏障来转发来源空间指针，所以其无法直接通过来源空间中的对象获取其在目标空间中的对应地址（因为复制式回收器并不会对来源空间对象中的指针进行转发）。我们将这一结论称为灰色赋值器来源空间不变式 (fromspace invariant)。当然，为确保回收周期能够结束，所有赋值器最终必须仅持有目标空间指针，因此，对于允许灰色赋值器操作来源空间对象的复制式回收器而言，其必须确保最终能够完成所有赋值器根的转发，从而将所有赋值器都翻转到目标空间。另外，赋值器在来源空间所执行的操作也必须在目标空间得到重放，否则这些更新将会丢失。

17.1 主体并发复制：Baker 算法

最简单的一种并发复制策略可能是为所有赋值器线程维护目标空间不变式，该策略可以确保赋值器不会访问到回收器尚未复制或者正在复制的对象。在主体并发复制的框架下，为满足目标空间不变式的要求，回收器必须在回收周期的开始阶段（原子化地）挂起所有赋值器线程，同时完成其根的采样和转发（即复制根的目标对象）。该阶段完成后，已经被着为黑色的赋值器将仅包含指向目标空间的（灰色）指针，而这些指针的灰色目标对象依然可能包含来源空间指针。Baker[1978] 的黑色赋值器读屏障最初是为增量回收而设计的，其目的在于阻止赋值器获取来源空间指针，Halstead [1985] 进一步将其扩展到并发复制场景。读屏障可以确保赋值器线程不会穿过来源空间和目标空间之间的回收波面，从而给赋值器线程造成了回收过程已经结束的假象。

算法 17.1 描述了 Baker 式并发回收算法，它是算法 4.2 所示的万物静止式复制回收算法的改进版本。需要注意的是，只有当赋值器从目标空间灰色对象中加载指针时（即所加载指

针的目标对象可能位于回收波面之前)才会触发读屏障,此时 forward 方法必须确保其所返回的引用位于目标空间中,如果必要,其可能需要对来源空间中的对象进行复制。该算法中赋值器和回收器之间的同步粒度相对较粗(对象级别),即回收器需要原子化地扫描灰色对象,而赋值器读屏障也需要原子化地转发从灰色对象中加载的引用。两个 **atomic** 代码段可以确保赋值器线程永远不会从正在扫描(复制)的对象中读取引用。

算法 17.1 主体并发复制

```

1  shared worklist  $\leftarrow$  empty
2
3  collect():
4      atomic
5          flip()
6          for each fld in Roots
7              process(fld)
8      loop
9          atomic
10             if isEmpty(worklist)
11                 break                                /* 退出循环 */
12             ref  $\leftarrow$  remove(worklist)
13             scan(ref)
14
15 flip():
16     fromspace, tospace  $\leftarrow$  tospace, fromspace
17     free, top  $\leftarrow$  tospace, tospace + extent
18
19 scan(toRef):
20     for each fld in Pointers(toRef)
21         process(fld)
22
23 process(fld):
24     fromRef  $\leftarrow$  *fld
25     if fromRef  $\neq$  null
26         *fld  $\leftarrow$  forward(fromRef)                /* 将其更新为目标空间的引用 */
27
28 forward(fromRef):
29     toRef  $\leftarrow$  forwardingAddress(fromRef)
30     if toRef = null                                /* 尚未得到复制(标记) */
31         toRef  $\leftarrow$  copy(fromRef)
32     return toRef
33
34 copy(fromRef):
35     toRef  $\leftarrow$  free
36     free  $\leftarrow$  free + size(fromRef)
37     if free > top
38         error "Out of memory"
39     move(fromRef, toRef)
40     forwardingAddress(fromRef)  $\leftarrow$  toRef            /* 标记 */
41     add(worklist, toRef)
42     return toRef
43
44 atomic Read(src, i):
45     ref  $\leftarrow$  src[i]
46     if isGrey(src)
47         ref  $\leftarrow$  forward(ref)
48     return ref

```

在算法 17.1 中, Read 方法的 **atomic** 前缀将确保赋值器能够正确地判断对象 src 的状态

(即是否为灰色)以及目标对象的状态(即是否已完成转发),同时也能确保赋值器在不干涉 process 方法中回收器复制行为的前提下自主完成来源空间中目标对象的复制。此时 Read 操作的额外开销与其所复制对象的大小成正比,如果更加仔细地设计读操作与回收器之间的同步机制,算法将达到更细粒度的同步级别。

获取细粒度同步方式的一种策略是在工作列表中记录域地址而非对象的引用,该方案的难点在于如何将灰色域与黑色域进行区分,即赋值器如何才能简单高效地判定某个域是否位于回收波面之前。当以对象作为最小着色单元时,回收器可以通过对象头部的一个位来标记其是否为灰色,而如果将域作为最小着色单元,使用相同的标记策略将会引入很大的额外存储开销。通过观察我们可以发现,在 Cheney 扫描中,指针 scan 会随着回收器逐个扫描每个域的过程而(原子化地)向前递进,因此,位于指针 scan 之后的域均为黑色,其之前的域均为灰色。我们可以基于这一结果设计新的读屏障:

```
atomic Read(src, i):
    ref ← src[i]
    if ref ≠ null && scan ≤ &src[i]                /* src[i]为灰色 */
        ref ← forward(ref)
    return ref
```

对于回收器如何在堆中以域为单位原子化地推进回收波面,我们将在第 19 章介绍具体的实现细节。届时我们将看到,该技术可以最大限度地减少回收器对赋值器的中断,这对于实时系统将是十分重要的。

主体并发、主体复制回收

主体并发回收算法天然适用于主体复制式回收器。主体并发回收器必须保守地对待每个模糊根(ambiguous root),即其必须钉住所有被模糊根引用的对象,但是,对于其他并未直接被模糊根引用的对象,回收器便可自由地将其移动。因此,主体并发回收器显然可以通过一个短暂的万物静止阶段来扫描线程栈和寄存器,同时标记(并钉住)所有被模糊根所引用的对象。该阶段完成后,所有赋值器线程将全为黑色,此时 Baker 式读屏障便可确保赋值器在后续执行过程中永远不会触达尚未完成复制的对象。

DeTreville [1990] 在 Modula-2+ 以及后续的 Modula-3 [Cardelli 等, 1992] 这两种面向系统的语言中使用了这一策略,这两种语言的编译器均无法生成准确的栈映射(stack map)信息,且编译器也无法在赋值器访问堆的操作中插入显式屏障,因而 DeTreville 使用了基于虚拟内存页保护机制的 Appel 等 [1988] 读屏障来实现赋值器与回收器之间的同步。Detlefs [1990] 在 C++ 中也使用了相同的技术,他对 AT&T C++ 编译器进行了修改,使得编译器可以自动生成堆中对象的精确指针映射信息,从而允许回收器对未被模糊根直接引用的对象进行复制。

在使用抢占式调度策略的操作系统中,回收器将很难实现页保护的自动管理,因而 Hosking [2006] 使用编译器生成的、对象粒度的读屏障来替代粒度较粗的虚拟内存保护机制。由于读屏障仅会在回收过程的复制阶段激活,且回收器在万物静止阶段会扫描模糊根并将其着为黑色,所以在读屏障检测来源对象是否为灰色的快速路径中便可避免昂贵的原子操作开销,此时只需保证读屏障在真正执行转发操作时的原子性即可满足要求[⊖]。

⊖ 由于黑色赋值器不可能持有白色引用,所以在算法 17.1 中,Read 操作的 src 参数只能为黑色或者灰色,此时即使去掉 if 语句的原子性,传递给 forward 方法的参数 ref 在最差情况下只是已经完成了转发而已,此时 forward 方法只需简单将其忽略即可。——译者注

17.2 Brooks 间接屏障

维护目标空间不变式的另一种策略是允许赋值器在执行过程中忽略回收波面的位置。Brooks[1984] 注意到, 如果每个对象 (不论其在来源空间还是目标空间) 都存在一个非空转发指针 (不论其指向来源空间中的原始对象, 还是目标空间中的新副本), 则读屏障中判断 `src` 对象是否为灰色的操作便可省略。来源空间对象在其得到复制之前, 其内部的转发指针域将指向其自身, 当其得到复制后, 转发指针域便会自动更新为目标空间中新副本的地址。对于对象在目标空间的新副本, 其转发指针域也将指向其自身。引入该屏障之后, 赋值器所有的堆访问操作 (包括指针、非指针、头部可变值的读/写操作) 都必须无条件地对间接指针域进行解引用, 如果目标空间存在该对象的新副本, 则赋值器将能够正确地访问到新对象。Brooks 式间接屏障如算法 17.2 所示。

算法 17.2 Brooks 式间接屏障

```

1  atomic Read(src, i):
2      src ← forwardingAddress(src)
3      return src[i]
4
5  atomic Write(src, i, ref):
6      src ← forwardingAddress(src)
7      if isBlack(src)                /* src 位于目标空间的回收波面之后 */
8          ref ← forward(ref)
9      src[i] ← ref

```

Brooks 间接屏障的唯一问题在于读屏障依然可能访问到回收波面之前尚未完成复制的对象, 但间接指针域的存在放宽了目标空间不变式的要求, 此时赋值器不仅可以操作灰色对象, 而且可以持有来源空间引用。为确保回收过程可以正常结束, Brooks 使用 Dijkstra 式写屏障来拦截赋值器将来源空间指针插入到回收波面之后的操作, 如算法 17.2 所示。

由于赋值器线程可以操作灰色对象, 所以一旦复制过程完成, 回收器需要再次扫描赋值器线程的栈, 并将其中所有尚未得到转发的引用替换。原始的增量式 Brooks 回收器所使用的是另一种策略, 即回收器在每个回收增量结束之后扫描赋值器线程的栈与寄存器, 并在恢复其执行之前完成所有待转发引用的重定向。

17.3 自删除读屏障

Baker 式回收器需要使用读屏障来维护黑色赋值器不变式。由于读操作通常比写操作更加普遍, 所以读屏障的开销通常会比写屏障昂贵得多。另外, 读屏障通常是条件性的, 即在 `Read(src, i)` 中, 读屏障必须检测 `src[i]` 是否已经位于目标空间, 如果结果为否, 则需将其复制。Cheadle 等 [2004] 提出了一种方案, 该方案可以消除赋值器在访问目标空间对象时的屏障开销, 并在 Glasgow Haskell Compiler (GHC) 的 Baker 式增量复制回收器中得到实际应用。GHC 中每个对象 (闭包) 的第一个字都指向其入口代码 (entry code), 即对该闭包进行求值的代码。在标准的闭包求值代码之外, 他们又引入了一种新的求值代码, 后者会在调用标准求值代码之前完成闭包的复制, 其具体执行流程如下: 在非回收周期内, 入口指针将指向不进行复制的标准代码; 在回收启动后, 如果某一对象被复制到目标空间, 则回收器会将新副本的入口指针修改为具有自我复制能力的求值代码。一旦赋值器访问到目标空间中的新对象并对其进行求值, 则求值代码首先会将该对象的子节点复制到目标空间, 然后再将

其入口指针恢复为标准版本，最后再执行标准的求值代码。该策略的巧妙之处在于，如果某一闭包会在未来进行求值，则赋值器必然要无条件执行其求值代码，而读屏障则可以在求值代码中删除。该策略仅会产生少量的重复代码开销，Cheadle 等发现这一开销仅比万物静止式复制回收器高 25%。他们还将这一技术应用于 Jikes RVM Java 虚拟机中，并通过劫持方法表指针的方式来实现屏障的自我删除 [Cheadle 等, 2008]，但为实现这一目的，对象的大多数访问操作（包括所有的方法调用、内部域的访问，`static` 变量与 `private` 变量除外）都必须虚拟化（virtualize）。为尽量降低这一策略所引入的开销，他们还使用了一种较为激进的技术，即通过运行时编译来将代码内联。

17.4 副本复制

Brooks 间接屏障在时间和空间两方面均存在额外开销：其不仅要求赋值器的每次堆访问操作（包括读 / 写指针 / 非指针）都对间接指针解引用，还需要在每个对象头部预留一个完整的字来保存间接指针。该屏障的优势在于，其不仅放宽了 Baker 式读屏障在加载来源空间引用时必须复制其目标对象的限制，而且可以确保赋值器所访问到的（包括读和写）一定是对象在目标空间中的新副本（如果其存在的话）。据此，我们可以得出一个重要结论：Brooks 间接屏障可以确保堆中对象的更新永远不会丢失，即更新操作要么发生在对象得到复制之前，要么发生在复制完成后目标空间的新副本上^①。

[341]

副本复制回收器 [Nettles 等, 1992 ; Nettles and O'Toole, 1993] 允许赋值器继续操作来源空间中对象的原始副本，即使回收器正在复制该对象，从而进一步放宽了 Brooks 间接屏障的要求。此时赋值器线程将遵守来源空间不变式，即赋值器可以直接更新来源空间中的对象，同时写屏障需要记录所有更新操作，且这些操作必须在目标空间的新副本中得到重放。也就是说，副本复制回收器允许目标空间中对象的状态落后于来源空间的原始对象，并且将这一状态保持到回收器完成复制之后，但是在回收来源空间之前，回收器必须确保所有针对来源空间对象的操作都在目标空间中得到重放，且所有赋值器根都已得到转发。因此，回收周期的结束条件将是赋值器操作日志为空，且所有赋值器的根、所有位于目标空间的对象都已得到扫描。

在并发副本复制算法中，赋值器和回收器在处理操作日志和更新赋值器根时都需要进行同步。在处理操作日志时，使用线程本地分配缓冲区、工作窃取技术均可以最大限度地降低同步开销 [Azagury 等, 1999]。回收器必须确保在回收结束之前所有对象与其副本的状态均保持一致。当回收器^②对某个已经扫描过的副本进行修改时，其必须重新扫描该副本，进而确保由该修改操作所产生的所有新引用都在目标空间中得到体现。为确保回收周期能够结束，回收器需要将所有赋值器挂起并扫描器根。当所有对象已完成扫描、操作日志为空且赋值器不再持有任何未复制对象的引用时，则意味着回收周期的结束。此时赋值器线程全部处于挂起状态，回收器便可安全地重定向其根，进而将赋值器线程全部翻转到目标空间。

该算法所产生的开销仅仅是短暂挂起赋值器以便对其根进行采样，并在回收结束时对其进行重定向：回收器可以逐个挂起每个线程并扫描其根，而在回收结束时，则需要在一个短暂的万物静止阶段中将所有线程翻转到目标空间。

① 尽管如此，但原子化地复制一个对象并写入转发地址，实现起来通常并不简单。

② 依照赋值器所生成的日志。——译者注

副本复制算法的不足之处在于,写屏障必须对赋值器在堆中的每一次更新操作进行记录,不论其所操作的数据是否为指针,这一开销通常会比传统的仅拦截指针操作的写屏障高得多,此时操作日志将很容易成为整个系统的瓶颈。但是对于不允许修改对象的函数式语言(例如 Nettles 和 O'Toole 的 ML),便不存在这一性能问题。

17.5 多版本复制

Nettles 和 O'Toole[1993] 的副本复制回收器依然需要使用万物静止的方式来将赋值器线程同步迁移到目标空间。由于在迁移过程中所有赋值器线程都无法继续执行,所以该算法无法满足无锁(lock-free)要求。Halstead[1985] 对 Baker[1978] 的算法进行了多处理器优化,其方案为每个处理器在堆中分配了独立的空间,即每个处理器都拥有专属的来源空间与目标空间。与此同时,不论处理器在扫描过程中所发现的存活对象位于哪个来源空间,其都有义务将该对象复制到自身目标空间中。Halstead 使用锁来解决多处理器在复制同一对象时可能产生的竞争问题,并且不允许赋值器更新正在进行复制的对象,他同时还需要使用万物静止式的方式来将所有处理器翻转到目标空间,然后才能将来源空间整体回收。为消除回收过程中的全局停顿,Herlihy 和 Moss[1992] 将回收来源空间与翻转处理器到目标空间这两步操作解耦。他们将每个处理器所对应的空间划分为一个目标空间以及多个(零个或者更多)来源空间。在复制过程中,同一对象的来源空间副本可以存在于多个处理器的空间中,但是在任意时刻只有一个副本能够成为当前版本,而其他副本都属于已废弃版本。

每个处理器[⊖]所扮演的角色将在赋值器和回收器之间变换。当其执行回收器的相关任务时会在局部变量以及目标空间中进行扫描,目的是寻找指向来源空间中已废弃版本的指针。当找到此类指针后,回收器会定位该对象的当前版本。如果当前版本位于来源空间中,则回收器会将其复制到自身目标空间并令其成为当前版本,来源空间中的旧版本将被废弃。

[342]

处理器将通过上述方式实现对象从来源空间到目标空间的复制,同时完成不可达指针向目标空间的重定向。每个处理器不仅要在自身目标空间中扫描来源空间指针,而且有义务将其所发现的所有当前版本位于来源空间的对象复制到自身目标空间中(包括位于其他处理器来源空间中的对象)。处理器可以在赋值器执行过程中的任意时刻执行来源空间和目标空间的翻转(通常是当目标空间已满导致无法分配新对象时),但不得在扫描过程中进行翻转。在某一处理器执行翻转之后,只有当其他处理器都不再持有其来源空间的引用时,该来源空间才能整体回收。

为了对同一对象的不同版本进行管理,Herlihy 和 Moss 需要在每个对象内部维护一个额外的转发指针域 next,每个已废弃版本的 next 指针将指向其下一个版本,从而形成了一条从最老版本到最新版本的指针链,链的末尾便是对象的当前版本,其 next 指针为 null。当处理器将某一对象复制到自身目标空间后,其需要使用 CompareAndSwap 操作原子化地将自身目标空间中新版本的地址写入指针链末尾对象的 next 指针域,从而令其成为当前版本。因此,赋值器在每次进行堆访问之前都必须沿着对象的版本指针链追溯到其当前版本。另外,为了在确保堆中更新操作不丢失的前提下满足无锁要求,赋值器在每次更新对象之前都必须先在处理器的目标空间中创建一个新版本,然后基于这一新版本进行更新,最后使用

[⊖] Herlihy 和 Moss 使用进程(process)这一术语,但我们将依照惯例使用“线程”这一概念。为了与 Halstead [1985] 的理念保持一致,并进一步强调“每堆区域”这一概念,我们将继续使用处理器(processor)这一术语。

CompareAndSwap 操作令其成为当前版本（即写时复制）。此时扫描操作与复制操作之间将无需引入任何全局性的同步，同时也可确保所有赋值器更新都不会丢失。

只有当所有执行扫描的处理器（即扫描者，scanner）都不再持有某一来源空间的引用时，其所对应的处理器（即所有者，owner）才能将该来源空间回收。对于所有者而言，如果任意一个扫描者都没有找到指向该所有者来源空间的指针，则称扫描的结果为干净（clean），否则其结果为脏（dirty）；每个处理器均完成开始扫描到结束扫描的整个过程，称为一轮（round）；如果某一轮扫描完成后所有处理器的扫描结果都为干净，且期间没有处理器执行翻转操作，则称该轮扫描的结果为干净。在某一处理器执行翻转操作之后，其来源空间必须等到下一轮结果为干净的扫描结束之后才能得到回收。

每个所有者都会使用两个原子握手位（handshake bit）来探测扫描者的启动与停止，每个位均是由一个处理器写而由另一个处理器读。在初始状态下，两个握手位的值相等。当所有者需要进行翻转时，其首先创建一个新的目标空间，然后将老的目标空间标记为来源空间，最后将所有者的握手位反转（invert）。当扫描者开始扫描某一所有者的空间时，其先读取该所有者的握手位，然后执行扫描，最后再将其之前读取到的值写回到该所有者的扫描者握手位中。因此，如果在扫描过程中所有者的握手位没有发生反转，则在扫描完成后两个握手位的值将依然相等。

每个所有者必须探测其他处理器扫描过程的开始和结束，同时由于每个处理器都可以扮演所有者和扫描者的角色，所以握手位的实现应当是两个数组：所有者数组仅包含所有者握手位，并以所有者的处理器编号作为索引；扫描者数组则需要通过二维数组的方式实现，数组的每个元素对应一个〈所有者，扫描者〉对的扫描者握手位。由于一次扫描会涉及多个所有者，所以扫描者在扫描之前必须先将整个所有者数组复制到本地，并且在扫描完成后将其之前读取到的每个值设置到扫描者数组对应的握手位中。如果某一所有者发现在扫描者数组中，所有扫描者所设置的握手位均与自身所有者的握手位相同，则意味着该轮扫描结束。在本轮扫描结束之前，任意所有者均不得执行翻转。

为检测已经完成的一轮扫描结果是否为干净，处理器之间将共享一个脏标记位数组，其中的每个位对应一个处理器。任意所有者在执行翻转之前都会将所有处理器对应的标记位设置为脏。与此同时，如果扫描者发现了指向其他处理器来源空间的指针，该处理器的脏标记也会得到设置。如果某一所有者的脏标记在一轮扫描结束之后处于干净状态，则称该所有者在该轮扫描的结果为干净，其便可以回收自身的来源空间；否则其将简单地清理自身的脏标记，并发起一轮新的扫描。另外，如果将脏标记与来源空间而非处理器相关联，则当扫描者发现位于某一来源空间的对象时便可仅为其设置脏标记，进而允许所有者独立回收每个来源空间。

Herlihy 和 Moss 证明了算法的安全性，但是他们并未针对算法的具体实现进行性能分析。确保算法安全性的基础在于，如果每个处理器最终都能完成扫描，则部分处理器终究可以回收来源空间。在最差情况下，每个处理器都将耗尽自身空闲内存，这将导致翻转操作无法进行，此时所有处理器都必须集中精力进行扫描，直到某一轮扫描的结果为干净为止。当然，一旦这种资源耗尽的情况发生，系统必然会被整体挂起。

避免写时复制的措施

多版本复制算法的亮点在于其整个处理过程完全无锁，但其缺陷也十分明显：在每次

堆更新操作时均需要创建一个新的版本（尽管在非一致内存结构的多处理器上该策略可能提升局部性）。针对这一问题，Herlihy 和 Moss 提出了多种策略来避免每次更新时的写时复制。

- **使用 CompareAndSwap2 进行原地更新。**该策略假定处理器支持 CompareAndSwap2 原语，该原语可以确保处理器的原地更新操作只有在转发指针 next 依然为 null 的情况下才会执行。但是，CompareAndSwap2 原语在现代处理器上并未得到广泛实现，事务内存（transaction memory）是一个可能的替代策略。事实上，该方案是启发 Herlihy 和 Moss 进一步改进措施的灵感。
- **所有者原地更新。**该策略只需依赖 CompareAndSwap 原语，但其需要在对象头部引入数个额外的域：例如对于对象 a 而言，seq(a) 表示其序列号模 2 的值，index(a) 表示待更新槽的索引号，value(a) 表示待写入的值。next(a) 代表转发指针域，该域既可用于保存转发地址（可以为 null），还可以额外持有一个序列号（其实现方式是：在对象满足一定对齐要求的前提下，可以在转发指针的低位增加额外的标签值）。对象序列号的值仅可能为 0 或者 1，如果 seq(a)=next(a)，则原地更新成功，否则更新将被忽略。

在执行写入之前，处理器必须沿着对象的版本链找到其当前版本（即当转发指针域为 null 或者为版本号时）。如果当前版本位于本地目标空间中，则处理器执行算法 17.3 所示的 WriteLocal 方法。该方法的传入参数为对象的当前版本 a、处理器所观察到的转发指针域 next（为 null 或者版本号）、待修改域的索引号 i 以及即将写入该域的值 v。该方法使用 CompareAndSwap 原语来为 next 域原子化地设置新版本号，如果成功，则处理器通过删除屏障来扫描即将被覆盖的指针（目的是维护“所有写入目标空间的指针都会得到扫描”这一不变式），然后再执行真正的写入操作。如果失败，则处理器需要重新找到其当前版本，并使用完整的写屏障来尝试对其进行更新。对于非一致内存架构的处理器而言，处理器更新本地对象的速度更快，因而所有者原地更新策略尤其适用于这一场景。

344

算法 17.3 Herlihy 和 Moss [1992] 的所有者原地更新写屏障

```
1 WriteLocal(a, next, i, v):
2   seq ← (next + 1) % 2
3   seq(a) ← seq
4   index(a) ← i
5   value(a) ← v
6   if CompareAndSet(&next(a), next, seq)
7     scan(a[i])
8     a[i] ← v
9   else
10    Write(a, i, v)
```

如果待更新对象 a 并非处于本地目标空间中，则新的所有者依然需要在本地目标空间中创建新副本，但在新的所有者完成复制之后且执行写入之前，新的所有者必须检查 next(a) 和 seq(a) 是否依然相等。如果相等，则所有者将首先写入本次要更新的值，然后通过删除屏障对 a 中 index 域所表示的槽进行扫描，最后再将 value(a) 中的值写入到其在当前版本所对应的槽中。扫描者在将对象迁移到自身目标空间时也需执行相同的操作。这一操作可以确保复制过程中所有发生在原有版本中的写操作可以在最新版本中线性化地执行。

- **基于锁的原地更新。**最后一种策略是放弃无锁保障，并在更新对象时使用 Compare-

AndSwap 为对象加锁。与其他策略相同, 依然只有当前版本的所有者才能进行原地更新, 其更新过程如下:

- 1) 使用 CompareAndSwap 原子化地将某一表示上锁的值写入 next 域。
- 2) 对即将被覆盖的指针进行扫描 (如果存在的话)。
- 3) 执行更新。
- 4) 对写入的指针进行扫描 (如果存在的话)。
- 5) 将 next 域改回 null, 实现解锁。

由于只有对象的所有者可以进行原地更新, 所以这一操作无需与扫描者进行同步。第 2) 步中的删除屏障可以确保即将被覆盖但在覆盖之前可能被其他处理器读取的指针得到扫描; 第 4) 步中的插入屏障可以确保即使该对象刚被扫描过, 新插入的指针也不会被错误地忽略。

17.6 Sapphire 回收器

Halstead [1985]、Herlihy 和 Moss [1992] 均采用将堆划分为多个对称区域, 且每个处理器负责回收一个区域的策略。该策略的问题之一在于堆结构必须与多处理器拓扑结构紧耦合, 实际应用中的堆结构以及线程级并行机制并不能很简单地适用这一模型。除此之外, 单个处理器也可能成为算法执行过程中的瓶颈: 每个处理器都负责堆中一块巨大的空间或者多块空间, 所以可能出现由于某一处理器未完成扫描而导致其他处理器都无法释放自身来源空间的情况, 进而导致赋值器因无法分配内存而产生停顿。我们可以考虑让剩余空间不足的处理器从其他处理器中窃取空闲内存, 但这便要求回收器能够在运行时动态地重新配置每个处理器所对应的堆空间, 此处的相关问题我们曾在第 14 章讨论过。本节我们所要介绍的是非并行 (non-parallel) 并发回收器, 此类回收器可以将回收工作非对称地指定给一个或者多个专门的回收线程, 且其可以通过调整回收线程优先级的方式来实现赋值器线程与回收器线程之间的吞吐量平衡。

[345]

Sapphire 回收器 [Hudson and Moss, 2001, 2003] 是一种并发复制式回收器, 该回收器主要是针对运行在配备中小规模共享内存的多处理器之上, 且赋值器线程数量较多的应用程序而设计的。该算法对 17.4 节所介绍的并发副本复制算法进行了改进, 其允许一次仅对一个赋值器线程进行翻转而不必同时将所有赋值器线程挂起, 并一次性地将它们全部迁移到目标空间, 从而最大限度地减少了由于垃圾回收而产生的停顿。当赋值器同时操作来源空间与目标空间时, 如果某一对象在两个空间中都存在副本, Sapphire 回收器要求赋值器必须对两个副本都进行更新。Sapphire 回收器在堆中单独开辟了一块新对象空间 (newspace) 来用于对象的分配, 新分配的对象将是黑色的, 它们必然能够在当前回收周期中得到保留。与目标空间不同, 回收器将不会对新对象空间进行扫描。引入新对象空间之后, 由于来源空间和目标空间的大小均存在上限, 所以有助于回收周期的结束。由于回收器会把新分配对象当作黑色 (即位于回收波面之后), 以避免对其进行扫描, 所以回收器必须引入写屏障来捕获向新分配对象中写入的指针, 并对其进行扫描以及转发。

17.6.1 回收的各个阶段

Sapphire 回收器的回收过程主要分为 MarkCopy 和 Flip 两大阶段 (如算法 17.4 所示)。

算法 17.4 Sapphire 回收器的主要回收阶段

```
1 MarkCopy:
2     Mark                                /* 标记可达对象 */
3     Allocate                            /* 分配目标空间外壳 */
4     Copy                                /* 将来源空间中的数据复制到目标空间外壳中 */
5
6 Mark:
7     PreMark                            /* 激活 Mark 阶段所要用到的写屏障 */
8     RootMark                          /* 标记并扫描全局变量，最终将其着为黑色 */
9     HeapMark/StackMark                /* 处理回收器标记队列 */
10
11 Flip:
12     PreFlip                            /* 激活 Flip 阶段所要用到的写屏障 */
13     HeapFlip                          /* 将堆中所有的来源空间指针翻转到目标空间 */
14     ThreadFlip                        /* 依次翻转每个线程 */
15     Reclaim                            /* 回收来源空间 */
```

1. MarkCopy

在该阶段中，回收器会标记直接从全局变量、赋值器线程栈、寄存器可达的对象，并将其复制到目标空间。在该阶段的处理过程中，赋值器的读操作将依然访问对象在来源空间中的原始副本，但其写操作必须同时镜像到其在目标空间的新副本中。为避免对非 volatile 域的读操作施加读屏障，两个空间中的副本通过编程语言的内存模型来保持松散一致性（对于 Java 而言，可以参见 [Manson 等，2005；Gosling 等，2005]，其同样适用于 C++ 未来的内存模型 [Boehm and Weiser, 1988][⊖]），这意味着对每个副本的更新不必是原子化的或者同步的。例如对于 Java 应用程序而言，确保两副本中的数据在应用程序级别的同步点（application-level synchronisation point）是一致的即可。也就是说，在程序执行到下一个同步点之前，赋值器线程在来源空间副本中的所有更新操作都将在目标空间的副本中得到重放。当所有线程都到达同步点时，来源空间与目标空间中的副本将彼此一致[⊖]。这一特征对于 Flip 阶段将是十分重要的，因为在 Flip 阶段中，赋值器将可以同时访问来源空间和目标空间中的副本。

346

2. Flip

在该阶段中，回收器会对全局变量、线程栈、寄存器中的指针进行转发，并逐个将每个线程翻转到目标空间。尚未翻转到目标空间的赋值器线程可能会同时持有两个空间中副本的引用（甚至可能是同一对象在两个空间中的副本）。对于本章前面所介绍的各种并发复制式回收器，它们要么使用读屏障来阻止赋值器访问来源空间，进而维护目标空间不变式 [Baker, 1978]，要么在复制的过程中维护来源空间不变式并一次性完成所有赋值器线程的翻转 [Nettles and O’Toole, 1993]。不使用读屏障的增量翻转技术意味着赋值器可以同时访问来源空间与目标空间中的对象，此时便需要确保同一对象的两个副本严格同步。

对于同一对象在两个空间中的两个副本，它们的引用在语言级别应当是相等的，因而这一规则会影响指针相等的判断逻辑。每个判断指针是否相等的操作必须执行算法 17.5a 所示的屏障。需要注意的是，只要两个参数中的任意一个为 null，编译器即可将判断函数简化为 p=q。与 Brooks[1984] 的屏障类似，Flip 阶段也必须使用 flipPointerEQ 函数（参见算法 17.5b）来比较已经得到转发的指针。

⊖ 即 C++11 所提供的内存模型。——译者注
⊖ 我们在此强调，Sapphire 回收器假定赋值器线程在对非 volatile 变量进行更新时不会产生竞争。

算法 17.5 Sapphire 回收器中的指针相等判断

(a) 快速路径

```

1 pointerEQ(p, q):
2     if p = q return true
3     if q = null return false
4     if p = null return false
5     return flipPointerEQ(p, q)      /* 仅在 Flip 阶段才会调用 */

```

(b) Flip 阶段的判断逻辑

```

1 flipPointerEQ(p, q):
2     pp ← forward(p)
3     qq ← forward(q)
4     return pp = qq

```

(c) 指针转发

```

1 forward(p):                                /* p 为非空指针 */
2     pp ← toAddress(p)    /* 如果 p 位于目标空间, 则 pp 将为空指针 */
3     if pp = null
4         pp ← p
5     return pp

```

- **MarkCopy: Mark.** Mark 阶段会对来源空间中每个从根（包括全局变量、线程栈 / 寄存器）可达的对象进行标记。在 Sapphire 回收算法中，回收器使用工作队列来处理所有的标记任务，而赋值器写屏障则负责将回收相关指针压入标记队列：当赋值器将某个对象的引用写入全局变量或者堆时，如果该对象位于来源空间且尚未得到标记，赋值器便需将其压入标记队列。回收器首先扫描全局变量，如果其发现指向来源空间中未标记对象的引用，则其需要将该引用加入工作队列。然后回收器将对标记队列中的引用进行标记和扫描：当其将对象 p 的引用从标记队列中移出时，如果 p 尚未得到标记，则对其进行标记，同时扫描其指针域，并将其所引用的、位于来源空间的未标记对象压入标记队列。

当回收器发现标记队列为空时，它会逐次挂起每个赋值器线程，扫描其栈、寄存器，并将新发现的、指向来源空间未标记对象的引用压入标记队列。如果回收器完成所有赋值器线程的扫描之后标记队列依然为空，则意味着标记结束，否则回收器必须进行标记与扫描。由于写屏障能够确保回收波面不会后退，且新分配的对象均为黑色，所以回收周期必然可以结束，来源空间中的所有可达对象最终必然都将得到标记与扫描。

Mark 阶段分为 3 个子阶段。PreMark 阶段会激活标记阶段所需的写屏障 $Write_{Mark}$ ，如算法 17.6a 所示。赋值器并不直接参与标记工作，其只需要将未标记对象的引用压入标记队列以便回收器进行处理。来源空间中的未标记对象为隐式白色对象，标记队列中的对象为隐式灰色对象，这一信息可以通过一个表示对象是否已经入队的位来进行编码，该位同时也可以避免将同一对象多次入队。每个赋值器线程都拥有本地标记队列，因而入队操作通常无需任何同步操作。当回收器扫描赋值器线程栈时，其会将赋值器线程标记队列中的元素全部转移到自身标记队列中。

在 RootMark 阶段，回收将扫描全局变量并使用已经激活的 $Write_{Mark}$ 屏障将它们所引用的、尚未标记的对象添加到标记队列中，从而将全局变量全部着为黑色。新分配对象也被视

为黑色，因而在其中执行写操作（包括初始化过程中的写操作）也会触发 `WriteMark` 屏障。此时赋值器栈以及寄存器依然为灰色。

最后，回收器将在 `HeapMark/StackMark` 阶段处理自身标记队列、赋值器线程栈，以及一个额外的显式灰色对象集合。对于标记队列中的每个引用，回收器首先判断其是否已被标记，如果没有，则对其进行标记，并将其添加到显式灰色对象集合，以便进一步扫描（已标记过的对象将被忽略）。回收器将对显式灰色集合中的每个对象进行扫描，并使用 `WriteMark` 屏障将其所引用的、尚未标记的对象添加到标记队列中，此时该对象将变成黑色，即已得到标记但并不在显式灰色对象集合中的对象为黑色。回收器会持续进行迭代，直到标记栈与显式灰色集合均变空为止。（同一对象可能会多次进入标记队列，但不论如何，其最终都将得到标记，并且不会再被赋值器压入标记队列。）

当标记队列以及灰色对象集合变空时，回收器会将某个赋值器线程短暂挂起在安全回收点（该点不可能位于写屏障的执行过程中），然后使用 `WriteMark` 方法扫描其栈、寄存器并将其着为黑色。如果回收器在扫描每个线程的栈 / 寄存器时均未发现白色指针（即未将任何对象压入标记队列），且标记队列与灰色对象集合皆为空，则意味着全局变量、线程栈 / 寄存器、新分配对象中不可能存在白色指针，即它们均为黑色。正是由于写屏障可以确保全局变量以及新分配对象均为黑色，回收器才能确保该阶段能够结束。写屏障能够阻止赋值器向堆中写入白色引用，因此赋值器获取白色指针的唯一方式便是从白色或灰色可达对象中加载引用。当回收器完成所有赋值器线程的扫描之后，堆中将不存在任何灰色对象，此时赋值器将只可能从白色对象中加载白色引用。但是由于赋值器在得到扫描之后便不可能持有任何白色引用，因此在扫描完成之后，赋值器便不可能再访问到白色对象。这一规则适用于所有赋值器，因而所有赋值器线程必然都为黑色。

348

需要注意的是，在 `Mark` 阶段，只有那些在得到扫描之后继续执行的赋值器线程才需要重新扫描，类似地，只有那些在得到扫描之后依然活动的栈才需要重新扫描。

该阶段完成后，来源空间中的所有白色对象均为不可达对象。

算法 17.6 Sapphire 回收器中的写屏障

(a)Mark 阶段的写屏障

```
1 WriteMark(p, i, q):
2   p[i] ← q
3   if isFromSpace(q) && not marked(q)           /* q 为白色 */
4     enqueue(q)                                   /* 回收器将在稍后对其进行标记 */
```

(b)Copy 阶段的写屏障

```
1 WriteCopy(p, i, q):
2   p[i] ← q                                     $
3   pp ← toAddress(p)                             $
4   if pp ≠ null                                  /* p 位于来源空间 */
5     q ← forward(q)                             /* 若 q 非指针，则可省略这一步 */
6   pp[i] ← q                                     $
```

(c)Flip 阶段的写屏障

```
1 WriteFlip(p, i, q):
2   q ← forward(q)                             /* 若 q 非指针，则可省略这一步 */
```

```

3   p[i] ← q
4   pp ← toAddress(p)
5   if pp ≠ null                      /* p 位于来源空间 */
6       pp[i] ← q
7       return
8   pp ← fromAddress(p)
9   if pp ≠ null                      /* p 位于目标空间 */
10      pp[i] ← q
11      return

```

- **MarkCopy : Allocate**。当标记阶段确定来源空间中的可达对象集合之后，回收器将为来源空间中的每个已标记对象创建空的目标空间外壳（shell）。回收器不仅会在来源空间对象中设置指向其目标空间副本的转发指针，而且会通过一张哈希表反向记录每个目标空间副本所对应的来源空间对象，其目的在于：当某个赋值器线程翻转到目标空间后，其他尚未完成翻转的线程依然会访问来源空间，因此该线程对目标空间中对象的操作依然需要同步到来源空间。
- **MarkCopy : Copy**。当回收器为来源空间中的每个已标记对象都创建了目标空间副本且设置了转发指针之后，其便可以开始将来源空间中对象的数据复制到其所对应的目标空间外壳中。该阶段需要遵从“目标空间中的对象只能持有目标空间中对象的引用”这一不变式，为达到这一目的，赋值器将使用 `WriteCopy` 这一新的写屏障，该屏障不仅可以确保一次写操作可以同时更新来源空间与目标空间中的两个副本，还可以确保写入目标空间的指针所引用的对象必然位于目标空间，如算法 17.6b 所示。由于所有赋值器线程都尚未翻转，即它们依然工作在来源空间中，所以此时写屏障的作用仅仅是将来源空间中对象的变更同步到其在目标空间的副本中。此处的 `toAddress` 方法与其他复制策略中的 `forwardingAddress` 方法含义相同，即如果该方法的传入参数本身就是指向目标空间的指针，则其返回值将为 `null`；而此处的 `forward` 方法则是将把来源空间指针转化为其所对应的目标空间指针，如果传入参数本身即为指向目标空间的指针，则简单地将该值返回。该屏障中的内存访问操作必须以代码所指定的顺序执行（带 `§` 标记的代码），否则该屏障将无法正确实现同步，因为 Sapphire 假定赋值器之间不会针对非 `volatile` 变量产生竞争。
- **Copy : Copy** 阶段由以下几个子阶段构成：子阶段 `PreCopy` 将设置 `Copy` 阶段所要用的写屏障 `WriteCopy`，如算法 17.6b 所示；子阶段 `Copy` 会将来源空间中的每个黑色（已标记且已完成扫描）对象复制到其在目标空间的外壳中。赋值器可能会在回收器复制对象的同时对其进行修改，为此，回收器需要使用算法 17.7 所示的无锁同步方式来解决可能产生的竞争。该算法首先尝试不借助同步原语将地址 `p` 的值复制到地址 `q`，并在发现 `p` 的值发生变化时进行有限次数的重试，如果重试到达上限，则使用 `LoadLinked/StoreConditionally`（LL/SC）原语来协助完成复制（由于在复制过程中，赋值器尚未翻转到目标空间，因而从 LL 到 SC 的一段时间内，如果 `q` 发生了变化，必然是赋值器在对 `p` 进行更新时通过写屏障更新了 `q`，此时即使 SC 操作失败，`p` 和 `q` 将依然保持一致）。需要再次强调的是，程序必须以代码所指明的顺序（带 `§` 标记的代码）来访问内存。另外，需要注意的是，算法假定赋值器对来源空间中地址 `p` 的更新（进而也更新 `q`）是导致 `StoreConditionally` 操作失败的唯一原因，但不幸的是，并发硬件通常无法提供这一保障（即 SC 操作可能产生假性失败），因此仅依

赖 LoadLinked/StoreConditionally 原语便不足以确保程序的正确性[⊖]。在实际应用中，对 copyWord 方法必须进行更加保守的改进，即 copyWordSafe 方法，但是如果赋值器在 LL 和 SC 操作之间不断更新地址 p 的值，则回收器便无法正常向前执行。

算法 17.7 Sapphire 回收器的复制过程

```
1  copyWord(p, q):
2      for i ← 1 to MAX_RETRY do
3          toValue ← *p
4          toValue ← forward(toValue) /* 若 toValue 非指针，则可省略这一步 */
5          *q ← toValue
6          fromValue ← *p
7          if toValue = fromValue
8              return
9      LoadLinked(q)
10     toValue ← *p
11     toValue ← forward(toValue) /* 若 toValue 非指针，则可省略这一步 */
12     StoreConditionally(q, toValue) /* 假设不出现假性失败 */
13
14 copyWordSafe(p, q):
15     for ...
16     loop
17         LoadLinked(q)
18         toValue ← *p
19         toValue ← forward(toValue) /* 若 toValue 非指针，则可省略这一步 */
20         if StoreConditionally(q, toValue)
21             return
```

- Flip：该阶段同样也分为数个子阶段。刚刚进入该阶段时，尚未翻转的赋值器可以同时操作来源空间与目标空间中的数据，因而 PreFlip 子阶段需要设置 Flip 阶段的写屏障 Write_{Flip} 以应对这一情况（见算法 17.6c）。HeapFlip 子阶段会将全局变量以及新生空间中所有来源空间指针翻转到目标空间。Write_{Flip} 可以确保只有目标空间指针才能写入全局变量或者新生空间，从而保证了回收工作不会出现倒退。ThreadFlip 子阶段会依次翻转每个赋值器线程，即：将线程挂起并对其栈、寄存器中的来源空间指针进行翻转，然后再恢复线程的执行。在该子阶段中，所有赋值器依然需要对来源空间和目标空间中的副本同时进行更新，因此已经翻转的线程需要能够找到目标空间任意对象在来源空间中的对应副本（即用与 toAddress 方法所对应的 fromAddress 方法）。最后，一旦所有赋值器都完成翻转，且所有赋值器线程都将不再执行 Write_{Flip} 写屏障，Reclaim 子阶段便可将来源空间整体回收，同时将记录目标空间到来源空间对应关系的反向映射表一并回收。

由于尚未完成翻转的线程依然可能同时访问来源空间和目标空间中同一对象的副本，所以判断指针是否相等的操作需要对目标空间指针进行比较（见算法 17.5b）。

17.6.2 相邻阶段的合并

Sapphire 回收器允许将某些阶段合并。例如 RootMark、HeapMark/StackMark、Allocate、Copy 可以合并成一个单独的 Replicate 阶段，同时将 Write_{Mark} 和 Write_{Copy} 合并成一个单独的 Replicate 写屏障：当赋值器将来源空间指针写入目标空间时，写屏障会将来源空间指针添加

⊖ 感谢 Laurence Hellyer 指出这一问题。

到某一队列，以便回收器对其进行复制，同时也将该指针所写入的域入队，以便回收器在复制完成后修正该域。

17.6.3 Volatile 域

Java 要求每次对 `volatile` 域的访问都必须真正访问物理内存，且多个处理器之间的并发访问必须表现出顺序一致性。因此，不仅赋值器在访问 `volatile` 域时需要执行较重的同步逻辑，回收器在对其进行复制时也需要确保其两个副本保持适当的一致性。Hudson 和 Moss 提出了多种方案来解决这一问题，每种方案均会给 `volatile` 域的访问引入显著的额外开销。

综上所述，Sapphire 回收器对以往的各种并发复制算法进行了扩展，并与副本复制算法具有较多的共同点。该算法不仅允许将赋值器线程逐个从来源空间翻转到目标空间（无需将其全部挂起），而且最大限度地减小了赋值器线程的停顿时间，同时在访问非 `volatile` 域时还可以省去读屏障开销。当同一对象在来源空间和目标空间中都存在副本时，赋值器对两个副本都进行更新，从而确保了两副本之间的一致性。

17.7 并发整理算法

我们曾在第 3 章介绍过整理式回收算法，此类算法的执行可以划分为两个阶段，即标记阶段与整理阶段。我们注意到，在整理式回收算法中，确定可达对象集合的追踪过程与整理过程相对独立，因而整理算法在对象的重排列顺序方面具有更高的自由度，也就是说，复制式回收只能以追踪过程的遍历顺序进行复制，而整理式回收则可按照地址顺序进行滑动整理。

351

17.7.1 Compressor 回收器

我们曾在 3.4 节以及 14.8 节提到过 Compressor 回收器 [Kerny and Petrank, 2006]，该回收器正是充分利用了整理式回收将标记过程与复制过程分离的更大自由性，从而实现了并发整理。

回顾 14.8 节可知，Compressor 回收器首先计算出一个辅助的首对象表（first-object table），该表所记录的内容是未来将会复制到某一目标空间页的第一个来源空间对象。然后，并行整理线程将通过竞争的方式获取尚未映射物理内存的目标空间虚拟内存页，为其映射物理内存页并将来源空间中的对象复制到该页，同时还需要将该页中的所有指针域重定向到目标空间的对应地址。当某一来源空间页中所有存活对象都得到复制之后，回收器会立即解除该页的内存映射（unmap）。

为支持并发整理，Compressor 回收器使用与 Appel 等 [1988] 类似的虚拟内存页保护策略，后者利用虚拟内存页保护策略来扮演读屏障的角色，从而阻止赋值器访问包含未复制对象或者未转发指针的页；Ossia 等 [2004] 也使用页保护策略来对包含已整理对象的页中的指针进行并发转发。Compressor 回收器的整理与转发均依赖页保护策略。在回收开始时，Compressor 回收器首先禁止赋值器对目标空间进行读写访问（即不为目标空间中的页映射物理内存页）；其次，计算首对象表并对赋值器线程并发访问的目标空间页进行保护^①；然

^① 在后文我们将看到，这些目标空间页是那些存活对象占比较高的内存页，回收器不必对其进行整理，但须对其中的引用进行转发。——译者注

后, Compressor 回收器短暂地挂起所有赋值器线程并将其根重定向到目标空间;最后再恢复赋值器线程的执行。该阶段完成后,目标空间尚不存在任何已复制对象,但在此时,一旦赋值器访问某一受保护的目标空间页,则将触发陷阱,陷阱处理函数会处理与该页相关的整理工作,即:为虚拟内存页映射物理内存、从来源空间复制对象(此时赋值器线程将扮演整理线程的角色,并承担了增量整理的工作)、对新复制对象中的引用进行转发,这些工作完成后,赋值器线程才能恢复执行并访问该页。需要注意的是,在这一过程中,赋值器线程仅完成了其所访问页中数据的复制,因而对于横跨当前页与上一页的对象,其前半部分将不会得到复制,类似地,对于横跨当前页与下一页的对象,其后半部分也不会得到复制。为支持并发整理,对于“不允许”赋值器线程访问的目标空间页,回收器的整理线程应当能够正常访问,因而整理线程在填充某个目标空间页之前必须使用二次映射(double-map)策略来映射物理内存页:不仅要原本的目标空间虚拟内存页映射到物理内存(但依然保留页保护策略),还要将某一尚未映射的、整理线程“私有”的虚拟内存页映射到相同的物理内存(可参见 11.10 节)。一旦整理线程完成该页的整理,便可解除目标空间虚拟内存页的保护策略,同时解除该过程所用到的“私有”映射。

从本质上讲,Compressor 回收器遵从标准的三色不变式,即来源空间页为白色、已被保护的目标空间页为灰色、已解除保护的目标空间页为黑色。在初始状态下,当回收器依照目标空间地址计算首对象表时,赋值器线程仅会对来源空间进行访问,因此其颜色为灰色;将赋值器线程翻转到目标空间,相当于是将其着为黑色,此时基于页保护策略的二次映射读屏障能够阻止赋值器线程访问正在由回收器线程填充的灰色目标空间页(从而进一步阻止黑色赋值器访问来源空间中的陈旧引用)。

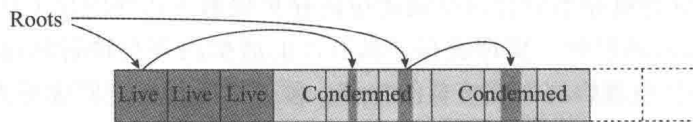
Compressor 回收器还必须处理其他与三色不变式相关的问题。特别是,在标记完成之后、开始计算首对象表之前,为避免赋值器新分配的对象干扰迁移映射表的计算,新对象必须分配在目标空间中(如果在来源空间的空洞中进行分配,则会造成干扰)。另外,当赋值器翻转到目标空间之后,回收器必须确保这些新分配对象最终能够得到扫描,并将其中的来源空间指针重定向到目标空间的正确副本,对于全局根也应如此。因此,回收器必须阻止赋值器访问目标空间中的新分配对象以及全局根,并在它们所处的页上设置陷阱,以强迫赋值器完成对这些对象中指针的扫描与重定向。

352

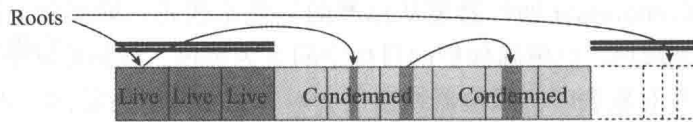
Compressor 回收器的性能严重依赖于虚拟内存页映射与保护操作的开销,由于这些操作的开销通常较大[Hosking and Moss, 1993],所以尽量将这些操作批量化便显得十分重要。例如在回收开始之前,回收器便会将目标空间作为整体进行页保护以及二次映射(对每一页分别执行这一操作将导致总开销过大)。基于相同的原因,Compressor 回收器的陷阱处理函数每次会解除 8 个虚拟内存页的保护(以达到陷阱开销与赋值器线程处理开销之间的平衡)。

Compressor 回收器的一个不足之处在于,一旦赋值器陷入目标空间的页保护陷阱,则其不但必须完成被保护页中所有对象的复制,还必须将这些对象中的所有指针转发到其目标对象迁移之后的地址(目标对象可能已完成迁移,也可能尚未完成),这便可能给赋值器带来显著的停顿。稍后我们将介绍 Pauseless 回收器,在该回收器中,赋值器在陷入页保护陷阱之后最多只需要完成一个对象的复制(且不需要对其中的任何来源空间指针进行转发),因而减少了赋值器线程的工作量。但在此之前,我们会首先简单回顾一下 Compressor 回收器基于页保护策略的整理算法,如图 17.1 所示,图中不同颜色的区域所代表的是虚拟内存页的逻辑分组(堆的线性地址布局并未在图中得到展示):

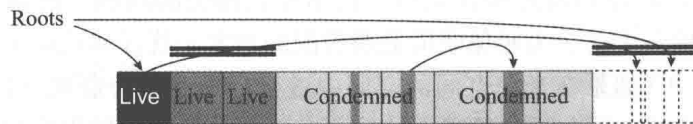
- **存活 (live)**: 内存页中的大部分对象均为存活对象 (图 17.1 中初始状态为深灰色的区域)。
- **已定罪 (condemned)**: 内存页中包含少量存活对象, 即大部分对象均已死亡, 此类内存页具有较高的整理价值 (图 17.1 中的浅灰色区域, 深灰色代表其中的存活对象)。
- **空闲 (free)**: 内存页当前处于空闲状态, 可以用于分配 (图 17.1 中带虚线边框的区域)。
- **新存活 (new live)**: 内存页中的待复制对象已完成内存分配, 但尚未完成复制 (图 17.1 中边框为虚线且以黑色斜线填充的区域)。
- **死亡 (dead)**: 内存页处于未映射状态, 一旦没有任何指针指向该页中的对象, 该页便可被回收 (即释放, 以便再次用于分配) (图 17.1 中以灰色阴影线填充的区域)。



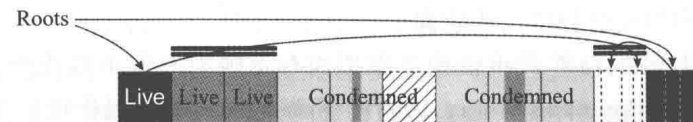
a) Compressor 回收器的初始状态, 此时所有内存页均位于来源空间



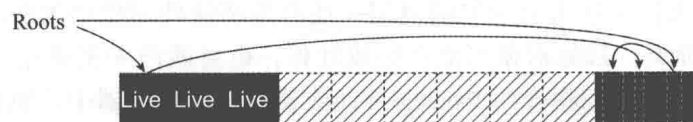
b) 计算转发信息, 并对所有目标空间页进行保护 (带双水平线标识的区域)。目标空间页包括未来用于容纳迁移对象的内存页以及并未定罪的存活页。然后, 赋值器的根将翻转到目标空间, 其访问目标空间受保护页的操作将会触发陷阱



c) 如果赋值器线程在访问存活页时触发陷阱, 则其必须将该页中所有的指针转发到目标空间的对应地址。完成存活页中所有来源空间指针的转发之后, 赋值器线程便可将其保护解除



d) 如果赋值器线程在访问目标空间的保留页时触发陷阱, 则其必须将来源空间对应页中的对象复制到该页, 同时必须完成其中指针域的转发。然后赋值器便可解除该目标空间页的保护, 同时将那些所有存活对象都已得到迁移的来源空间页解除映射 (释放其所对应的物理内存页, 如图中被灰色阴影线填充的区域所示)



e) 当所有存活页都已得到扫描 (其中的所有指针域都已得到转发), 且所有位于定罪页中的存活对象都已复制到目标空间 (且其所有指针域都已得到转发) 时, 整理过程结束

图 17.1 Compressor 回收器

图 17.1a 展示了回收开始阶段堆的状态, 此时回收器已经识别出所有存活对象, 并已经确定哪些对象需要迁移。为简化后续对 Pauseless 回收器的描述, 我们冒昧地限定回收器仅对存活对象较为稀疏的内存页进行整理。在 Compressor 回收器中, 回收器必须首先对包含来源空间指针的存活目标空间页、用于容纳迁移对象的目标空间页进行保护, 目的是避免赋值器线程直接访问它们。在赋值器线程执行的同时, 回收器线程会计算存活对象的转发信息, 并将其记录在辅助数据结构中。此时, 堆中内存页的布局将如图 17.1b 所示。然后回收器会将所有赋值器线程的根翻转到目标空间, 它们所引用的全部内存页都将处于保护状态。接下来回收器线程便可开始并发整理, 期间, 一旦有赋值器线程访问到尚未解除保护的目标空间页, 则会触发陷阱。如果赋值器线程在访问存活目标空间页时触发陷阱, 则必须将该页中的所有引用转发到目标空间, 如图 17.1c 所示; 如果赋值器线程在访问目标空间中尚未填充对象的页时触发陷阱, 则必须使用对应的定罪来源空间页来填充该页, 且在填充完成之后还需要对该页中的引用进行转发 (见图 17.1d)。当某一定罪来源空间页中的所有对象都已完成迁移之后, 该页的状态将变为死亡, 系统便可解除其物理内存映射, 并将其所对应的物理内存页归还给操作系统, 但是其虚拟内存页依然要保留到堆中所有引用都已得到转发为止。当所有目标空间页都已得到处理且都已解除映射时 (见图 17.1e), 便标志着整理过程的结束。接下来我们将对 Compressor 回收器与 Pauseless 进行比较。

353
?
354

17.7.2 Pauseless 回收器

Compressor 回收器需要对目标空间中的保留页或者包含来源空间指针的页进行保护, 而 Pauseless 回收器 [Click 等, 2005; Azul, 2008] 及其分代改进版本 C4 回收器 [Tene 等, 2011] 则是对来源空间中的待整理页进行保护。因此它无需像 Compressor 回收器那样对所有目标空间页进行保护, 而是只需保护真正包含需要迁移的对象的页 (即集中精力回收那些存活对象分布较为稀疏的、整理价值较高的页), 这不仅可以减少需要保护的页数量, 而且回收器也可对待整理页进行增量式的保护。Pauseless 回收器使用读屏障来拦截赋值器访问来源空间引用的操作, 并在赋值器载入该引用之前将其转发, 同时会避免赋值器对整个页中的来源空间指针进行修正。Pauseless 回收器的最初版本基于专属硬件系统来实现, 该硬件支持通过特殊的引用加载指令来直接实现读屏障, 但是对于一般的硬件系统, Pauseless 回收器需要将读屏障相关逻辑内联到赋值器的每个指针加载操作中。

硬件与操作系统支持。Azul 的专属硬件系统支持多种快速用户态陷阱处理函数。硬件的转译后备缓冲区除了支持传统的用户态与内核态这两种权限级别, 还额外支持第三种权限级别, 即 GC 态。某些快速用户态陷阱处理函数会切换到 GC 态执行。其转译后备缓冲区同时还支持大页表 (每页的大小为 1MB 或 2MB), 这通常要比标准处理器的传统页要大得多。大页是 Pauseless 回收器的标准工作单元。

该硬件系统还支持一种快速协作抢占 (co-operative preemption) 机制, 为支持这一机制, 处理器能够将中断位置限定在用户指定的指令上, 而这些指令通常与安全回收点 (GC-safe point) 相对应。也就是说, 被挂起的线程必然处于安全回收点, 因此处理器可以快速驱使运行中的线程执行到安全回收点, 然后再恢复该线程的执行, 整个过程无需引入等待。基于这一机制, 回收器可以引入一种十分快速的检查点 (checkpoint) 操作, 即当赋值器线程执行到检查点时, 回收器可以令其执行很少一部分的回收相关工作, 然后再恢复其正常执行, 而对于已经阻塞的线程, 其回收相关工作则由回收器代劳。相反, 在传统的万物静止式

回收器中，回收器只有确保所有赋值器线程都已到达安全回收点之后才能开始处理。运行中的线程在检查点处无需进行等待，回收工作也可在时间上得到均摊。

上文提到，Azul 的专属硬件系统支持硬件读屏障，该屏障可以根据回收器所处的特殊阶段执行多种检查与操作。算法 17.8 展示了该读屏障的主要逻辑：处理器首先执行用户指定的数据加载操作，然后再执行读屏障相关逻辑。读屏障通过转译后备缓冲区判断载入的值是否可能是一个地址。如果该“地址”（读屏障保守地假定其是一个地址）所在的页处于 GC 保护态，处理器便会激活快速用户态 GC 陷阱处理函数。该屏障会忽略空引用。与 Brooks 式间接屏障的区别之处在于，该屏障无需进行空指针判断、无需进行内存访问、不存在载入-使用（load-use）惩罚、无需在每个对象头部保留转发指针域，且不会增大高速缓存开销。

算法 17.8 Pauseless 回收器的读屏障

```

1 Read(src, i):
2     ref ← src[i]
3     if protected(ref)
4         ref ← GCtrap(ref, &src[i])
5     return ref
6
7 GCtrap(oldRef, addr):
8     newRef ← forward(oldRef)           /* 必要时进行转发 / 复制 */
9     mark(newRef)                       /* 必要时进行标记 */
10    loop                               /* 只有当 CAS 操作出现假性失败时，才进行重试 */
11        if oldRef = CompareAndSwap(addr, oldRef, newRef)
12            return                      /* CAS 操作成功，结束 */
13        if oldRef ≠ *addr
14            return                      /* 其他线程更新了 addr，但 newRef 依然可用 */

```

Pauseless 回收器需要从 64 位的指针中窃取一位用作回收相关标记，硬件在加载与存储引用时会忽略（过滤）该位。该位被称作“尚未标记过”（Not-Marked-Through, NMT）位，在并发标记阶段，赋值器通过该位来判定指针是否已经被回收器扫描过。硬件本身维护一个回收器所期望的 NMT 值，对于赋值器尝试载入的指针，如果其 NMT 值与该值不符，则赋值器线程将陷入 NMT 陷阱。该陷阱依然会忽略空指针。

在标准的硬件系统之上，读屏障只能通过软件方式实现，因而必然会引入额外开销。GC 保护态检查可以通过标准的页保护机制实现，而读屏障则可以通过无用加载指令（dead load instruction）或者显式查表的方式替代。NMT 位的检查可以通过内存多次映射（multi-map）的方式实现，即不同的页保护策略对应不同的 NMT 位。空引用通常在程序中十分普遍，因而必须对其进行显式过滤，编译器也可将 NMT 位的检查整合到编程语言（如 Java）现有的空指针安全检查中。软件过滤 NMT 位则需对编译器进行修改，其必须在赋值器的每处解引用操作之前过滤掉目标地址的 NMT 位，得到过滤的指针可以复用到下一个回收点[⊖]。除此之外，也可对操作系统进行修改，以便直接支持多映射内存或者地址区间别名，此时赋值器便可简单地忽略 NMT 位。

Pauseless 回收器的回收阶段。Pauseless 回收器的回收过程分为 3 个主要阶段，每个阶段都完全是并行且并发的。

⊖ 到下一个回收点之后，如果系统的 NMT 位依然没有发生变化，则可以继续复用。——译者注

- **标记 (mark)** 阶段需要周期性地刷新标记位。该过程需要将所有引用的 NMT 位设置到回收器所期望的值，同时需要对每一页的存活性进行静态分析。标记器从根（静态变量、全局变量、赋值器栈）开始标记可达对象。借助于指针中的 NMT 位，标记器可以与赋值器完全并发执行，我们将在稍后详述这一过程。
- **迁移 (relocate)** 阶段首先使用最新的标记位信息来查找堆中存活对象较为稀疏的页，然后对这些页进行整理（将其中的对象迁移），最后再释放其物理内存。

该阶段首先会对标记阶段选定的稀疏页进行保护，以阻止赋值器对其进行访问，然后再将其中的存活对象复制到其他页。回收器可以根据额外维护的转发信息来引导对象的迁移。如果赋值器加载了受保护页中的引用，则读屏障便会触发 GC 陷阱，该陷阱会将赋值器读取到的来源空间引用修正到已经得到正确转发的引用。当该页中的所有存活对象都完成迁移后，回收器便可释放其物理内存并立即将其归还给操作系统，而该页的虚拟内存则必须保留到其不再被任何指针引用时才能释放。

迁移阶段的执行是连续性的，其释放内存的过程可以与赋值器分配内存的操作同步进行。该阶段同时还可以与下一轮回收的标记阶段并发进行。

356

- **重映射 (remap)** 阶段会更新堆中所有指向已迁移对象的指针。

回收线程会遍历对象图，并会为堆中的每个引用执行读屏障，从而完成所有来源空间引用的转发，就像赋值器线程陷入陷阱所执行的那样。该阶段完成之后，堆中的任何对象都不再可能引用迁移阶段被保护的内存页，因此回收器可以将其虚拟地址空间释放。

由于重映射阶段与标记阶段都会对所有存活对象进行遍历，所以 **Pauseless** 回收器可以将它们合并，即本轮回收的重映射阶段可以与下一轮回收的标记阶段并发执行。

Pauseless 回收器存在诸多优势。首先，回收器无需十分急促地完成任何阶段。每个阶段都不会给赋值器带来显著的负担，因而每个阶段都没有快速结束的必要。在启动新一轮回收之前，回收器并不会强迫上一轮回收尽快结束——迁移阶段可以持续执行，且可以在任意时刻立即释放内存。由于所有阶段都是并行的，所以回收器可以通过调整回收线程数量的方式简单地匹配赋值器线程的分配速度。与其他并发标记回收器不同，不论赋值器的操作频率有多高，该回收器只需一次堆遍历便可确保完成标记过程（即对象不需要经历被标记、回退到灰色、再次被标记这一过程，也不需要借助于最终的万物静止方式来确保标记阶段的结束）。回收器线程将会与赋值器线程一起竞争 CPU 时间，它也可以当仁不让地占用赋值器线程所让出的 CPU 时间。

第二，该回收器还存在一种“自我修复”能力，即：当赋值器由于加载了某个引用而陷入读屏障陷阱时，陷阱处理函数可以立即对触发陷阱的引用进行更新，从而避免赋值器在同一位置再次触发陷阱。这一过程的开销取决于陷阱的类型。一旦赋值器的工作集合都已得到修复，它便可以全速运行而不会再陷入任何陷阱。这也导致了赋值器使用率在一个很短的时间内（即“陷阱风暴”时期）有所下降，最小的赋值器使用率会在 20ms 到数百毫秒的时间内受到影响。但无论如何，**Pauseless** 回收器也不会引入将所有赋值器线程同步挂起的万物静止阶段。下面我们将更加详细地描述各阶段的处理流程。

标记。标记阶段所使用的是位图标记策略。每个对象对应两个标记位，一个用于当前回收周期的标记，另一个则用于上一轮回收周期的标记。在标记阶段开始时，回收器首先将当前回收阶段的所有标记位清零，然后再对所有的全局引用进行标记（扫描每个赋值器线程根集合），扫描完一个线程之后，回收器将翻转该线程的期望 NMT 值。运行中的赋值器线程会

在回收检查点完成自身根集合的标记，而被阻塞（或者被挂起）的线程则会由回收器线程进行并行标记。赋值器线程可以在其根集合标记完毕（且期望 NMT 值得到置换）之后继续向前执行，但不论如何，只有当所有线程都通过检查点之后，回收器才能进一步处理标记阶段的其他工作。

完成根集合的标记之后，回收器便可使用类似于 Flood 等 [2001] 的方式进行并行标记，且整个过程可以与赋值器并发执行。指针中的 NMT 位仅对赋值器有效，标记器会将其忽略。回收器继续执行这一过程，直到所有存活对象都得到标记为止。新生对象将会在存活页中分配。由于赋值器只可能持有（以及写入）已经标记过的引用，所以新分配对象的初始标记位不会影响到标记过程。

在赋值器并发执行的情况下，回收器要想通过单次遍历完成所有存活对象的标记，NMT 位至关重要，因为读屏障需要依赖该位来阻止赋值器加载未标记对象的引用。当赋值器所加载引用的 NMT 值与回收器所期望的 NMT 值不符时，其便会陷入 NMT 陷阱，陷阱处理函数会协助标记线程完成该引用的标记。由于赋值器线程永远不会载入未标记对象的引用，所以也永远不可能将其写入其他位置。NMT 陷阱同时也会将已经修正（标记）过的引用写回内存，因此在后续过程中这一引用便不会再次引发陷阱。这一自我修复机制意味着在进入标记阶段之后，赋值器不必等待回收器完成其工作集合中对象内部指针域的 NMT 位的翻转，相反，赋值器线程会在运行时翻转其所遇到的每个引用的 NMT 位。稳态 NMT 陷阱通常很少。

[357]

当两个赋值器线程在同一地址触发 NMT 陷阱时，两个线程均会陷入陷阱处理函数并同时触发陷阱的引用进行更新，进而有可能导致冲突的产生。因此，陷阱处理函数有必要使用 CompareAndSwap 操作来更新引用，该操作可以确保陷阱处理函数仅在引用没有发生变化的前提下才将其更新。在标记阶段的开始，由于所有线程不可能同时到达检查点，所以各赋值器线程可能会在很短的一瞬间观察到不同的期望 NMT 值，因此两个线程的读屏障有可能会在同一个引用的 NMT 值上产生重复性的竞争。但这一局面在尚未翻转的线程到达下一个安全回收点（检查点）时便可结束，线程将在该点触发陷阱，然后标记自身栈，最后通过该检查点继续向前执行。

需要注意的是，同一线程的根集合不可能同时包含两个 NMT 位不同但所引用对象相同的指针，因而判断指针是否相等依然可以使用传统的逐位比较方式。

对于标记阶段的结束，唯一需要注意的问题是标记阶段的结束不应当发生在赋值器线程读取到一个未标记引用之后、为其执行读屏障之前。由于读屏障永远不可能跨越安全回收点，所以我们可以借助于安全回收点来达到这一要求。因此，标记阶段需要引入一个空的检查点，赋值器线程在该检查点之前需要照常对所发现的引用进行标记，如果在所有赋值器线程都通过检查点之后依然没有发现新的未标记引用，则标记阶段可以安全结束，否则标记线程需要对新发现的引用进行处理并设置下一个新的检查点。由于新创建的对象不可能包含错误的 NMT 位，所以标记阶段最终必然可以结束。

迁移。在迁移阶段首先需要找到存活对象较为稀疏的内存页。图 12.7 以不同的颜色展示了虚拟内存页的逻辑分组（我们再次强调，此处并未依照地址顺序来描述堆布局）。待整理内存页中的存活对象可能会被赋值器根以及其他存活页引用，因而在迁移阶段，回收器需要先构建一个额外的数组来保存待迁移对象的转发指针。由于待整理页所对应的物理内存会在其中存活对象迁移完成之后立即释放，且此时已迁移对象的来源引用并未全部得到转发，所以转发指针不能记录在待迁移对象中。回收器仅需对存活对象较为稀疏的页进行迁移，因

而用于记录转发指针的额外数组不会太大，且可以使用哈希表的方式实现。转发信息构建完毕之后，回收器便可修改定罪内存页的保护策略，从而阻止赋值器对其进行访问，如图 17.2b 所示。定罪页中的对象都应被看作是陈旧对象，赋值器不应再对其进行修改。当赋值器尝试加载某一指向定罪页的引用时，其将陷入 GC 陷阱并由读屏障进一步进行处理。

在定罪页受到保护之时，正在执行的赋值器很有可能已经持有了陈旧引用。回收器并不会直接对这些引用进行处理，而是由赋值器在通过检查点时对自身根集中的陈旧引用进行转发，并在必要时迁移其在来源空间中的目标对象（如图 17.2c 所示）。当所有赋值器线程都通过该检查点之后，回收器便可开始将剩余存活对象复制到目标空间，且这一工作可以与赋值器并发执行。读屏障可以阻止赋值器访问到尚未完成迁移的陈旧对象。

358

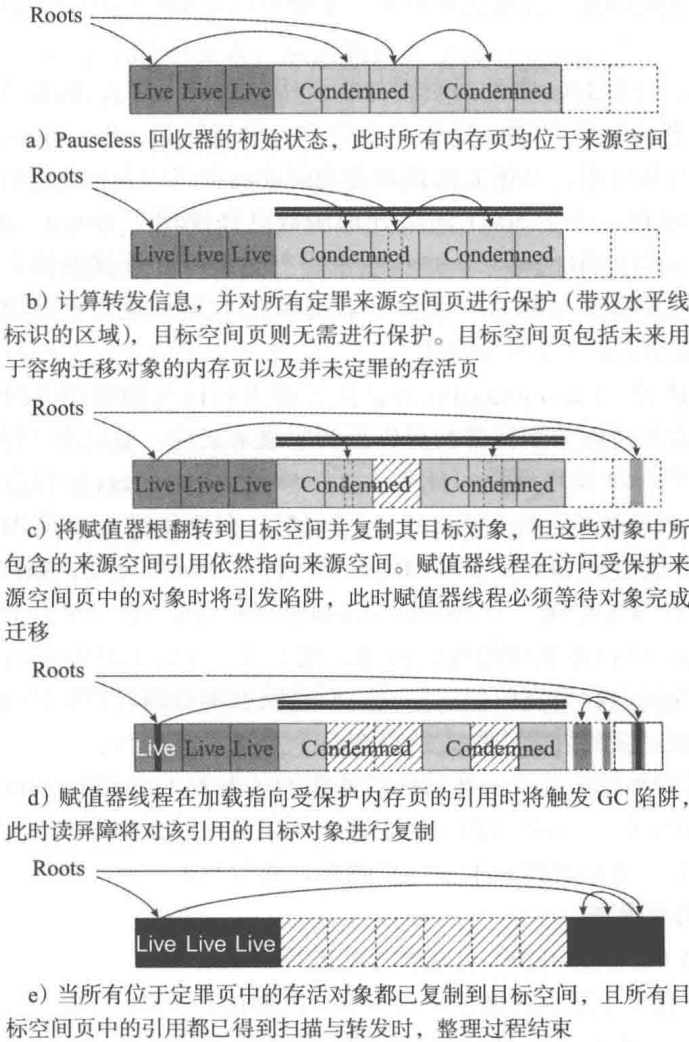


图 17.2 Pauseless 回收器

与标记阶段类似，在迁移阶段，阻止赋值器加载陈旧引用仍是读屏障的主要任务。具有自我修复能力的 GC 陷阱会对陈旧引用进行转发，然后再使用 CompareAndSwap 操作更新内存中的引用。如果来源空间对象尚未得到复制，则赋值器线程必须替代回收器完成复制，如图 17.2d 所示。由于 GC 陷阱处理函数运行在 GC 保护态，所以赋值器可以在这一状态下访问已受到 GC 保护的内存页。回收器不会对跨越多个内存页的对象进行迁移，也不会对存活对象占

比较高的内存页进行整理。大小达到半页的对象可以在大约 1ms 的时间内完成复制^①。

为分摊在转译后备缓冲区中修改保护策略的开销,以及对赋值器根进行转发的开销,Pauseless 回收器可以对存活对象较为稀疏的内存页进行分批整理,通常是每次对数 GB 的内存页进行保护(并迁移其中的存活对象,最后将其释放)。回收器只需确保其迁移速度能够与赋值器的分配速度相匹配即可。

重映射。对于存活对象已经迁移完毕的定罪页,其虚拟内存地址不能立即释放,因此回收器需要引入重映射阶段来对存活页中剩余的陈旧引用进行转发,并对来源空间页的虚拟内存进行回收。当重映射阶段结束时,存活页中将不存在任何指向来源空间页的指针,此时回收器便可将后者的虚拟内存回收(如图 17.2e 所示),额外的转发指针数组也可得到回收,一个完整的回收周期就此结束。需要注意的是,来源空间页的物理内存存在迁移阶段就已经得到回收。

终结与弱引用。对于 Java 中的弱引用与软引用(见 12.1 节),回收器将某一引用置空的操作可能会与赋值器读取该引用的操作产生竞争。幸运的是,Pauseless 回收器可以与赋值器并发处理弱引用与软引用,具体方法是在要求回收器在将尚未标记过的引用置空时必须使用 CompareAndSwap 操作。由于 NMT 陷阱处理函数已经使用了 CompareAndSwap 操作,所以赋值器与回收器都可以使用 CompareAndSwap 来进行竞争:如果赋值器在竞争中胜出,则引用将得到保留(回收器将感知到这一事实),而如果回收器在竞争中胜出,则引用便被成功置空(此时赋值器将只能读取到空引用)。

对操作系统的改进。Pauseless 回收器会较为激进且持久地使用虚拟内存映射与物理内存映射操作,这些操作可以使用标准的操作系统原语来实现,但是出于性能方面的考虑,如此高频度地使用这些标准操作原语可能令人无法接受。Pauseless 回收器对操作系统内存管理器的改造可以带来显著的性能提升[Azul, 2010]。在企业级 Java 应用程序中,每个处理器核心的内存分配率会达到 200 ~ 500MB/s,为匹配这一分配速度,系统必须能够达到较为持续的垃圾回收率以避免停顿。在 Pauseless 回收器中,每一页最终都将被重映射一次(然后再解除映射一次),以回收其中的死亡对象,这一过程不存在任何物理上的内存复制,因此重映射率并不会受到内存带宽的显著影响,而重映射本身的开销则是重映射率的主要决定因素。一般操作系统对重映射操作的支持存在 3 个限制:

1) 每次重映射操作都会隐含一次转译后备缓冲区失效(invalidation)操作。由于这一操作会引发多个(针对所有核心的)跨 CPU 中断,所以其开销会随着系统中活动线程的数量的增加而增大。即使某一活动线程并未参与重映射,或者与需要进行重映射的内存没有任何关联,也会受到该操作的影响。

2) 只有较小的(通常是 4KB)页映射才能进行重映射。

3) 重映射操作在单个进程内属于串行操作(需要持有一个写锁)。

为解决这些问题,Pauseless 回收器针对操作系统进行了一些改进,具体包括:重映射时无需将转译后备缓冲区失效(在完成大量内存页的重映射操作之后再行进行必要的失效操作)、支持大内存页(通常为 2MB)的重映射,以及同一进程内多个重映射操作的并发。与传统的操作系统相比,这些改进措施可以带来大约三个数量级的速度提升,与此同时,当活动线程的数量翻倍时,重映射操作整体开销的增长也呈线性趋势。

① 要注意的是, Pauseless 回收器所使用的内存页是相当大的。

综上所述, Pauseless 回收器是专门为大型多处理器系统设计的、完全并行、完全并发的垃圾回收器, 它不需要引入万物静止的停顿, 而且可以在回收周期的任意时刻回收死亡对象, 因此回收器也无需急迫地完成某一阶段以免赋值器耗尽内存。某些回收阶段的切换可能会导致赋值器陷入“陷阱风暴”, 进而出现短暂的使用率下降, 但它的自我修复能力可以确保赋值器使用率能够快恢复到正常水平。

17.8 需要考虑的问题

本章介绍了如何通过并发复制回收、并发整理回收来同时达到降低内存碎片和减少赋值器停顿的目的。不论是哪种并发回收算法, 回收器都必须确保赋值器的操作不会导致对象丢失, 而对于移动式并发回收器而言, 回收器还必须确保赋值器不会访问到陈旧对象。某些算法为赋值器维护目标空间不变式, 从而确保其永远不可能持有来源空间陈旧对象的引用 [Baker, 1978]; 某些算法会引导赋值器访问目标空间中的新对象 (如果存在的话), 同时依然允许赋值器对来源空间进行操作 [Brooks, 1984]; 还有一些算法允许赋值器继续操作来源空间, 但需要确保来源空间中的每一步操作都必须在目标空间中得到重放 [Nettles 等, 1992; Nettles and O'Toole, 1993], 一旦所有存活对象都已完成复制, 回收器将通过一步独立的操作使所有赋值器翻转到目标空间。如果要避免这一全局翻转操作, 则必须将每个对象的多个版本链接起来, 而赋值器在访问对象时, 则必须通过遍历来找到其最新副本 [Herlihy and Moss, 1992]。另外, 如果允许赋值器同时对来源空间和目标空间中的副本进行更新, 则回收器可以对赋值器线程进行逐一翻转 [Hudson and Moss, 2001, 2003]。并发整理算法可以通过类似的方式实现, 但其不需要在每次回收过程中都对所有的存活对象进行复制 [Kerny and Petrank, 2006; Click 等, 2005; Azul, 2008]。

移动式并发回收器可能会比非移动式并发回收器产生更长的停顿时间: 赋值器的每次堆访问操作都可能需要等待某一对象 (或者某些对象) 完成复制, 或者需要重定向到对象的当前版本。Baker[1992] 设计出 Treadmill 算法来解决其最初的复制式回收器 [Baker, 1978] 中存在的问题。尽管复制式与整理式算法对避免碎片化问题而言十分必要, 但是对那些对停顿时间或频率较为敏感的应用程序而言, 它们可能无法满足要求。此类应用程序通常还可能工作于内存受限的环境中, 例如嵌入式系统, 此时避免堆内存的碎片化就显得更加重要。我们将在第 19 章介绍如何在停顿时间严格受限的环境中使用并发复制与并发整理回收。

并发引用计数算法

我们曾在第 5 章介绍过引用计数算法。传统引用计数算法主要存在两个问题：一是无法回收环状垃圾，二是开销过高，特别是在多赋值器线程并发执行的环境下。引用计数算法解决环状垃圾的方案是试验删除（trial deletion），即部分追踪。延迟引用计数可以避免赋值器对局部变量执行引用计数操作，合并引用计数可以避免许多可以彼此抵消的“冗余”引用计数操作。合并引用计数还有一个额外的副作用，即它能够容忍赋值器之间的竞争。上述三种解决方案都需要引入万物静止式的停顿，以便回收器修正引用计数、回收垃圾对象，本章我们将介绍如何放宽这一要求，以及如何对这些算法进行改进，从而允许引用计数回收线程与赋值器线程并发执行。

18.1 简单引用计数算法回顾

为确保回收过程的正确性，引用计数算法必须遵守“对象的引用计数值等于该对象的被引用次数”这一不变式。在多赋值器线程环境下，维护这一不变式将更加复杂。依靠直觉，安全地实现 Write 操作会比安全实现 Read 操作更加困难。更新一个指针域需要三步操作：首先增加新目标对象的引用计数，然后再减少老目标对象的引用计数，最后再执行指针写入操作。确保这三步操作的执行顺序是十分重要的，但即使满足这一要求，在多赋值器并发操作的情况下仍有可能出现问题[⊖]。对象既不能被过早地回收（例如，由于其引用计数临时性地掉零而被错误地回收），也不应当成为永久性的浮动垃圾。图 18.1 描述了这一问题：即使所有的引用计数加减操作都得到原子化的执行，线程之间的交替执行仍有可能产生错误的结果，因为，在此处，老目标对象的引用计数可能会减少两次，而某个新目标对象的引用计数则可能超过其真正被引用的次数。

线程 1 Write(o,i,x)	线程 2 Write(o,i,y)
addReference(x)	addReference(y)
old ← o[i]	old ← o[i]
deleteReference(old)	deleteReference(old)
o[i] ← x	o[i] ← y

图 18.1 在引用计数算法中，引用计数操作必须与指针更新保持同步。此处的两个线程同时更新同一个指针域，old 为每个线程 Write 方法的局部变量

并发引用计数算法的设计难点不仅仅在于引用计数域的增减，如果仅需要考虑这一问题，则用第 13 章所介绍的各种原子操作（例如 AtomicIncrement）便可轻松解决问题。更困难的问题在于如何保持引用计数变更操作与指针读写操作的同步。在算法 5.1 中，我们只是简单地指出赋值器的 Read 和 Write 操作都必须是原子化的，最简单的实现策略是对赋值器正在操作的对象（即 src 对象）加锁，如算法 18.1 所示。该算法显然满足安全性要求：当 Read 操作对 src 加锁之后，第 i 个域的值便不可能再发生变化；如果该值为空，Read 操作

⊖ 需要注意的是，我们并不关注产生竞争时用户程序的执行是否正确，但无论如何，我们必须保证堆的一致性。

显然是安全的；如果该值为某一目标对象 `tgt` 的引用，则在 `Read` 操作对 `src` 解锁之前，`tgt` 将至少存在一个引用，因而引用计数不变式可以确保 `tgt` 的引用计数不会掉零。因此，我们便可确保 `tgt` 对象不可能在 `Read` 过程中被释放，且 `addReference` 操作所更新的必然是目标对象的引用，而不是已经被释放掉的内存。类似地，`Write` 操作的安全性也能得到保障。

算法 18.1 基于锁的立即引用计数

```
1 Read(src, i):
2     lock(src)
3     tgt ← src[i]
4     addReference(tgt)
5     unlock(src)
6     return tgt
7
8 Write(src, i, ref):
9     addReference(ref)
10    lock(src)
11    old ← src[i]
12    src[i] ← ref
13    deleteReference(old)
14    unlock(src)
```

一种十分诱人的解决方案是基于通用的原子操作原语开发一种无锁解决方案，但不幸的是，针对单一内存地址的原子操作并不足以确保算法的安全性。`Write` 操作的无锁化方案并不存在问题。我们使用原子自增与自减操作来更新引用计数，同时使用 `CompareAndSwap` 操作来执行指针写入，如算法 18.2 所示。如果 `ref` 非空，则执行写操作的线程会持有 `ref` 的引用，因而在 `Write` 操作返回之前，`ref` 不可能被回收（不论是使用立即引用计数，还是延迟引用计数）。`Write` 操作将会通过自旋的方式尝试写入指针，直到 `CompareAndSet` 操作返回成功为止。只有写入操作成功的线程才会对正确的 `old` 目标对象执行引用计数减少操作，因而在 `deleteReference(old)` 被调用之前，目标对象 `old` 的引用计数将处于高估状态，从而不可能被过早回收。

然而，相同的策略却不能应用于 `Read` 操作：即使 `Read` 方法使用原子操作原语来更新引用计数，但在线程载入引用（第 19 行）和增加引用计数（第 20 行）这两步操作之间，依然可能会有其他线程删除指针 `src[i]`，并将其目标对象回收，除非我们在这一过程中对 `src` 加锁。增加引用计数的操作可能最终会施加到已经释放的，甚至已经被重新分配出去的内存上。

364

针对单一内存地址的原子操作并不足以构建完整的解决方案，为此，Detlefs 等 [2001, 2002b] 基于 13.3 节所介绍的 `CompareAndSwap2` 原语来解决这一问题。`CompareAndSwap2` 原语可以原子化地更新两个独立内存地址的值。尽管该策略并不足以在任何情况下都保持精确的引用计数，但它却可以满足一个较弱的不变式：①只要存在指向某一对象的指针，该对象的引用计数值便不可能掉零；②如果某一对象不再被任何指针引用，则其引用计数最终必然会降为零。算法 18.3 使用 `CompareAndSwap2` 原语来确保赋值器只有在指针域的目标对象没有发生变化的前提下才会增加其引用计数，从而可以避免错误地修改已经释放的对象。

算法 18.2 基于 `CompareAndSwap` 原语的立即引用计数存在问题

```
1 Write(src, i, ref):
2     if ref ≠ null
```

```

3      AtomicIncrement(&rc(ref))          /* 确保 ref 不会被释放 */
4      loop
5          old ← src[i]
6          if CompareAndSet(&src[i], old, ref)
7              deleteReference(old)
8          return
9
10 deleteReference(ref):                  /* 确保 ref 不为空, 或者不会被释放 */
11     if ref ≠ null
12         AtomicDecrement(&rc(ref))
13         if rc(ref) = 0
14             for each fld in Pointers(ref)
15                 deleteReference(*fld)
16             free(ref)
17
18 Read(src, i):
19     tgt = src[i]
20     AtomicIncrement(&rc(tgt))          /* 可能会出错! */
21     return tgt

```

算法 18.3 基于 CompareAndSwap2 的立即引用计数

```

1 Read(src, i, root):
2     loop
3         tgt ← src[i]
4         if tgt = null
5             return null
6         rc ← rc(tgt)
7         if CompareAndSet2(&src[i], &rc(tgt), tgt, rc, tgt, rc+1)
8             return tgt

```

365

18.2 缓冲引用计数

立即引用计数策略要么需要使用锁, 要么需要依赖 (目前) 尚未得到广泛支持的多内存地址原子操作原语。5.3 节所介绍的延迟引用计数会避免对局部变量施加引用计数操作, 同时会延迟零引用对象的回收, 从而在一定程度上解决了引用计数变更操作与指针读写操作的同步问题。但是对于如何减少将指针写入对象域时的引用计数变更开销, 延迟引用计数却无能为力。接下来我们将介绍缓冲引用计数策略, 该策略仅需在赋值器写屏障中使用简单的加载与写入操作, 同时其也支持多线程应用程序。

为避免多赋值器线程同时变更引用计数的同步开销, DeTreville[1990] 将每个指针更新操作的新旧引用写入日志中 (该技术应用在一个针对 Modula-2+ 的混合式回收器中, 该回收器使用标记-清扫回收算法作为处理环状垃圾的后备手段)。回收器使用一个单独的引用计数线程来处理日志并修正对象的引用计数值, 从而天然保证了引用计数变更操作的原子化。为避免可能存在的引用计数先减后增操作 (从而导致对象的过早回收), 引用计数线程会先执行引用计数增加操作, 然后再执行减少操作。不幸的是, 带缓冲的更新策略并未解决引用计数变更与指针写入操作之间的一致性问题。为此 DeTreville 提出了两种解决方案, 但它们均不能完全满足要求。第一种方案是为整个 Write 操作加锁, 该方案不仅可以确保更新日志能正确地添加到缓冲区中, 而且可以确保多线程更新时的同步要求。为了避免在每个写操作中引入锁, 其第二种方案是为每个赋值器线程配置本地缓冲区, 同时周期性地将缓冲区中的

数据传递给引用计数线程，但这又要求开发者必须小心地确保每一步指针写操作的原子性，并且在必要时通过手工加锁的方式来避免图 18.1 所示的问题出现。

Bacon 和 Rajan[2001] 也为每个赋值器线程配置了本地缓冲区，但其要求对指针域的更新必须为原子化的，如算法 18.4 所示，基于 CompareAndSwap 的操作以及重试可以满足这一要求。赋值器写屏障会将槽 i 的新旧引用记录到本地的 `myUpdate` 缓冲区中（第 9 行）。与此同时，他们还会将局部变量的引用计数变更操作延迟，并且将程序的执行粗略划分为一个个时段（epoch），回收器会维护一个全局时段计数器，同时每个线程也会维护它的本地时段计数器，系统基于非协同时段（ragged epoch）机制来确保对象不会被过早回收[⊖]。与延迟引用计数类似，处理器会周期性地中断某一赋值器线程的执行，扫描其栈并将所有被发现的引用添加到本地缓冲区 `myStackBuffer` 中，然后再将其 `myStackBuffer` 以及 `myUpdates` 中的数据传递给回收器，同时增加本地时段计数器的值 e 。最后，处理器会先将回收线程调度到下一个处理器，再恢复被中断线程的执行。

在第 k 轮回收中，回收线程最终会被调度到最后一个处理器上。回收器会先执行第 k 轮回收的引用计数增加操作，然后再执行第 $k-1$ 轮回收的引用计数减少操作，最后再增加全局时段计数器的值（为简化算法，我们假定算法 18.4 中的全局 `updatesBuffers` 数组无限大）。该算法的优势在于，回收器永远不需要在同一时刻将所有赋值器线程挂起，因而其属于即时回收算法。需要注意的是，算法所使用的是延迟引用计数的一个变种：在回收周期开始时，直接被（当前时段的）线程栈所引用的对象引用计数将会增加；而当回收周期结束时，引用计数得到减少的则是所有在上一个回收周期中被线程栈直接引用的对象。

算法 18.4 并发缓存引用计数回收

```

1  shared epoch
2  shared updatesBuffer[ ]           /* 每个时段使用一个缓冲区 */
3
4  Write(src, i, ref):
5      if src = Roots
6          src[i] ← ref
7      else
8          old ← AtomicExchange(&src[i], ref)
9          log(old, ref)
10
11 log(old, new):
12     myUpdates ← myUpdates + [{old, new}]
13
14 collect():
15     /* 每个处理器会将本地缓冲区中的数据追加到全局 updatesBuffer 中 */
16     myStackBuffer ← []
17     for each local ref in myStacks      /* 被延迟的引用计数 */
18         myStackBuffer ← myStackBuffer + [{ref, ref}]
19     atomic
20         updatesBuffer[e] ← updatesBuffer[e] + myStackBuffer
21     atomic
22         updatesBuffer[e] ← updatesBuffer[e] + myUpdates
23     myUpdates ← []
24     e ← e + 1
25

```

⊖ 所谓非协同时段，是指系统允许各线程处于不同的时段，而不是要求先到达某一时段的线程等待其他线程，19.6 节对此会有更加详细的描述。——译者注。

```

26     me ← myProcessorId
27     if me < MAX_PROCESSORS
28         schedule(collect, me+1) /* 将 collect() 调度到下一个处理器上 */
29     else
30         /* 最后一个处理器将负责更新引用计数 */
31         for each (old, new) in updatesBuffer[epoch]
32             addReference(new)
33         for each (old, new) in updatesBuffer[epoch-1]
34             deleteReference(old)
35         release(updatesBuffer[epoch-1]) /* 释放旧的缓冲区 */
36         epoch ← epoch + 1

```

18.3 并发环境下的环状引用计数处理

接下来，我们将考虑引用计数算法如何在不引入万物静止式停顿的前提下实现环状垃圾的回收。Recycler 回收器 [Bacon 等, 2001; Bacon and Rajan, 2001] 通过在堆中追踪备选子图的方式来回收环状垃圾，具体方案是对引用计数进行试验删除（trial deletion）。尽管缓冲引用计数策略可以将引用计数相关操作集中到一个空闲的处理器上，但 Recycler 回收器在并发环境中依然会面临如下 3 个问题：

- 1) 在 Recycler 回收器探测环状垃圾的过程中，由于赋值器会并发修改对象图，所以回收器很可能根本无法再次扫描同一个子图。
- 2) 试验删除可能会导致子图与存活对象图失去联系。
- 3) 引用计数可能会过时。

为解决上述 3 个问题，异步 Recycler 回收器的处理过程需要划分为两个阶段。第一阶段的执行过程与第 5 章所介绍的同步回收器几乎相同，但是对于 collectWhite 方法（见算法 5.5）所发现的对象，异步模式的回收器不是立即将其回收，而是将其保留到下一个回收周期，并只有在确定该对象在下一个回收周期依然是垃圾时才能将其回收。该方案存在诸多缺陷：首先，从理论上讲（实践中可能并非如此），可能会存在某些环状垃圾无法得到回收的情况，即回收器的完整性无法得到保障；其次，试验删除不能使用已有的引用计数值，它还要求在对象头部增加一个额外的环状引用计数域；再次，为避免错误地回收存活对象，回收器在第二阶段需要再次对备选环状垃圾进行追踪；最后，由于回收器必须修正不正确遍历所遗留的白色或者灰色对象的颜色，所以增加了引用计数写屏障的额外开销。

Recycler 回收器的根本问题在于，其试图将一种原本为同步环境而设计的回收算法应用在对象图不断发生变化的异步环境中。接下来，我们将介绍如何使用堆的固定快照来对 Recycler 回收器进行改进。

18.4 堆快照的获取

我们曾在第 5 章介绍过，合并引用计数会为回收器提供堆的快照。赋值器会将对象在（其某一指针域）被修改之前的副本通过线程本地缓冲区同步传递给回收器。在回收周期的开始阶段，回收器会挂起每个赋值器线程并获取其本地缓冲区中的数据，然后再为其分配新的缓冲区。回收器会通过对象（被修改之前）的副本找到其原本所引用的对象并减少其引用计数，然后再通过对象的当前版本找到其当前所引用的对象并增加其引用计数。在回收结束之前，所有对象的脏标记都将得到清理。

我们首先介绍如何让引用计数线程与赋值器线程并发执行（需要一个短暂的停顿来传递

缓冲区), 然后再考虑如何将这一并发算法进一步改进为即时算法。在第一种情况下, 回收器可以临时性地挂起所有赋值器线程, 并获取其缓冲区中的数据, 然后再恢复赋值器线程的执行。接下来, 回收器便需要修正每个已修改对象新旧子节点的引用计数。减少引用计数可以使用与同步环境相同的方式, 即根据日志中所记录的副本, 但处理引用计数的增加则稍为复杂 (见算法 18.5)。此时引用计数线程必须使用赋值器将日志传递给回收器这一时刻的对象状态来增加引用计数。由于在这一时刻之后赋值器可能会再次修改日志中所记录的对象, 因而此处需要考虑两种情况。

368

算法 18.5 基于滑动视图来更新引用计数

```

1  incrementNew(entry):                                /* 使用回收器日志中的数据 */
2      obj ← objFromLog(entry)                          /* 当前对象 */
3      if not dirty(obj)
4          replica ← copy(obj)                          /* 复制对象的所有指针域 */
5          if dirty(obj)
6              replica ← getLogPointer(obj) /* 当前时段的初始状态位于某个线程的日志中 */
7      else
8          replica ← getLogPointer(obj)
9
10     for each fld in Pointers(replica)
11         child ← *fld
12         if child ≠ null
13             rc(child) ← rc(child) + 1
14             mark(child)                                /* 如果需要对年轻代进行追踪 */

```

第一, 如果对象的状态依然为干净, 表示其状态并未发生变化, 此时引用计数线程便可直接增加对象当前子节点的引用计数。需要注意的是, 算法 18.5 中的 `incrementNew` 方法在获取干净对象的副本之后, 必须再次判断它的状态, 因为在创建副本的过程中赋值器线程可能会重新设置其脏标记。

第二, 如果对象在日志传递完成之后发生了变化, 则其状态必然为脏, 且其在变脏之前的状态必然已经被记录到某个赋值器线程新的日志缓冲区中, 与此同时, 对象的脏指针也会指向该缓冲区, 且回收器在访问数据时无需与任意赋值器线程进行同步。以图 18.2 为例, 对象 A 在当前时段得到更新 (`Write(A, 0, D)` 这一操作), 会导致其在上一时段结束时刻指向对象 C 的指针域被覆盖。

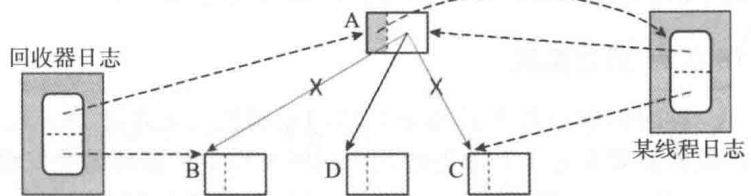


图 18.2 并发合并引用计数: 在上一时段, A 的第 0 个指针域从 B 更新到 C, 原有目标对象 (即 B) 的引用将被记录到日志中。但在当前时段, 该指针域又被更新到对象 D, 因而其重新被标记为脏, 同时再次被记录到日志中。与图 5.2 类似, 最初的引用 B 可以从回收器全局日志中找到, 而对象 C 的引用则可以从某个赋值器线程的当前日志中找到 (即对象 A 的 `getLogPointer` 域所引用的日志条目)

由于此时 A 的状态为脏, 则其在本时段首次被修改之前的状态必然已经记录在某个线程的本地日志中 (即图 18.2 中右侧的日志), 该日志记录了从 A 到 C 的引用。据此, 回收器便可减少对象 B 的引用计数, 并增加对象 C 的引用计数。而在下一个时段, 回收器则会根据 `Write(A, 0, D)` 的操作日志减少对象 C 的引用计数。

18.5 滑动视图引用计数

为获取堆的快照，我们需要以万物静止的方式挂起所有赋值器线程，然后再将其本地缓冲区中的操作日志传递给回收器。接下来我们将介绍如何放宽这一限制，从而达到一次只挂起一个赋值器线程的即时回收要求。在这一滑动视图 (sliding view) 方案中，不同对象的指针域将在不同的时刻得到记录，同时也会在不同的时间点传递给回收器线程，因此我们只能获取到堆的“扭曲”视图。滑动视图既不需要锁，也不需要任何原子操作指令（假定处理器满足顺序一致性要求），但是每个赋值器线程都需要与回收器线程进行 4 次握手，其握手的方式与 Doligez 和 Gonthier[1994] 类似。对于算法在弱一致性模型下所需做出的修改，我们将在稍后再做介绍。滑动视图可以应用于多种环境中，包括朴素引用计数 [Levanoni and Petrank, 1999, 2001, 2006]、分代回收器中年老代的管理 [Azatchi and Petrank, 2003]、面向年龄的回收器 [Paz 等, 2003, 2005b]，同时也可集成到环状引用计数回收器中 [Paz 等, 2005a, 2007]。此处我们仅考虑如何在面向年龄的回收器中使用滑动视图，以及如何对其进行扩展，以便能够回收环状垃圾。

369

18.5.1 面向年龄的回收

面向年龄 (age-oriented) 的回收器会将堆划分为年轻代和年老代。与传统的分代回收器不同，此处的年轻代对象和年老代对象将同时得到回收，因此不存在独立的年轻代对象回收，也不存在任何需要记录的分代间指针。回收器可以选择适当的策略来管理每个分代。弱分代假说告诉我们，大多数对象都将在年轻时死亡，而年轻对象也通常拥有更高的变更率（例如其初始化过程），因此，回收器在管理年轻代对象时可以充分利用其存活率低的特性，而在管理年老代对象时则可充分利用其死亡率低、变更率低的特性。Paz 等 [2003] 使用标记-清扫策略来管理年轻代对象（因为其不需要对大量死亡对象进行追踪），同时使用滑动视图引用计数策略来管理年老代对象（因为其可以处理包含大量存活对象的堆）。他们的面向年龄回收器并不会移动对象，而是通过对象的头部的一位来记录其所属的分代。

18.5.2 算法实现

即时回收过程首选需要获取滑动视图（见算法 18.6）。在对滑动视图进行增量回收时，回收器需要十分小心地处理获取视图过程中赋值器的并发修改操作。此时回收器需要为算法 5.3 中的 Write 操作增加一个额外的增量更新写屏障，该写屏障的作用是对写入的引用进行窥探 (snoop)，如算法 18.7 所示。对于仅被一个指针引用的对象 o ，窥探屏障可以确保如下情况下对象 o 不会丢失：某个赋值器线程先将其唯一引用该对象的指针域 s_1 覆盖，然后回收线程开始获取滑动视图，而当另一个指针域 s_2 被添加到滑动视图之后，赋值器线程又将对象 o 的引用写入 s_2 中。

算法 18.6 基于滑动视图的回收器

```

1  shared updates
2  shared snoopFlag[MAX_PROCESSORS]           /* 每个处理器对应一个标记 */
3
4  collect():
5      collectSlidingView()
6      即时握手 (4):
7          for each thread t
8              suspend(t)

```

```

9         scanStack(t)
10        snoopFlag[t] ← false
11        resume(t)
12    processReferenceCounts()
13    markNursery()
14    sweepNursery()
15    sweepZCT()
16    collectCycles()
17
18    collectSlidingView():
19        on-the-fly handshake 1:
20            for each thread t
21                suspend(t)
22                snoopFlag[t] ← true
23                transfer t's buffers to updates
24                resume(t)
25        clean modified and young objects
26        on-the-fly handshake 2:
27            for each thread t
28                suspend(t)
29                find modify-clean conflicts⊖
30                resume(t)
31        reinforce dirty objects
32        on-the-fly handshake 3:
33            for each thread t
34                suspend(t)
35                resume(t)
36
37    processReferenceCounts():
38        for each obj in updates
39            decrementOld(obj)
40            incrementNew(obj)
41
42    collectCycles():
43        markCandidates()
44        markLiveBlack()
45        scan()
46        collectWhite()
47        processBuffers()

```

算法 18.7 基于滑动视图的回收器: Write

```

1  shared logs[MAX_PROCESSORS]                /* 每处理器变量 */
2  shared snoopFlag[MAX_PROCESSORS]           /* 每处理器变量 */
3  me ← myProcessorId
4
5  Write(src, i, ref):
6      if src = Roots
7          src[i] ← ref
8      else
9          if not dirty(src)
10             log(src)                        $
11             src[i] ← ref                    $
12             snoop(ref)                      /* 窥探 */
13
14  log(ref):
15      for each fld in Pointers(ref)

```

⊖ 即找出在清理过程中赋值器所修改的对象。——译者注

```

16     if *fld ≠ null
17         add(logs[me], *fld)
18     if not dirty(ref)
19         /* 如果 ref 依然干净, 则提交日志 */
20         entry ← add(logs[me], ref)
21         logPointer(ref) ← entry
22
23 snoop(ref):
24     if snoopFlag[me] && ref ≠ null
25         mySnoopedBuffer ← mySnoopedBuffer + [ref]    /* 着为灰色 */

```

在回收周期开始时, 回收器将设置每个赋值器线程的 snoopFlag (该操作无需引入同步机制)。在回收器收集某一线程滑动视图的过程中 (此时, 该线程的 snoopFlag 必然已被设置), 任何由该线程写入堆中对象的引用都将添加到该线程的本地 mySnoopedBuffer 中 (见算法 18.7 中的第 25 行)。依照三色抽象的概念, 这一 Dijkstra 式写屏障将把 ref 着为黑色。为避免在初始化新分配对象的指针槽时触发写屏障, 算法 18.8 将新分配的年轻代对象着为灰色。

算法 18.8 基于滑动视图的回收器: New

```

1 New():
2     ref ← allocate()
3     add(myYoungSet, ref)
4     setDirty(ref)                                /* 将新分配的对象着为黑色 */
5     return ref

```

首先, 当回收器完成每个赋值器 snoopFlag 的设定之后, 其将发起第一次握手过程, 该过程会依次挂起每个赋值器线程, 并将其本地日志与年轻集合 (即算法 18.8 中的 myYoungSet) 传递到回收器的 updates 缓冲区中。

然后, 回收器将清理所有已修改对象以及年轻对象的脏标记。这一过程可能产生竞争: 清理过程是与赋值器并发执行的, 回收器可能会将赋值器刚刚修改的对象的脏标记清理。因此, 回收器需要与每个线程进行第二次即时握手, 仍然是依次挂起每个线程, 扫描其本地日志、识别出在清理过程中被修改的对象, 同时恢复这些对象的脏标记。

最后, 回收器引入一次空的握手来确保所有线程都已完成上一步操作。此时, 回收器便可开始对年轻代对象进行标记, 同时更新年老年代对象的引用计数。

在执行并发标记之前, 回收器先要发起第四次握手, 该过程仍然需要依次挂起每个赋值器线程, 依照传统的方式扫描其栈, 以便进行后续标记 - 清扫回收和延迟引用计数变更。该过程完成之后, 回收器便可清理每个线程的 snoopFlag。

每个赋值器线程的 mySnoopedBuffer 将会异步地传递到回收器的工作列表。此时回收器已经可以处理年老年代对象的引用计数。需要注意的是, 在年老年代对象处理完成之前, 回收器不应当对年轻代对象进行标记, 因为赋值器对年老年代对象的并发更新很有可能创建从年老年代对象到年轻代对象的引用。

回收器在对年老年代对象进行处理时需要依赖 updates 缓冲区中新老对象的引用计数。如果年轻代对象的引用计数得到更新 (说明该对象从年老年代对象可达, 其将被提升到年老年代), 且该对象尚未得到标记, 则其将被添加到标记器的工作列表中。

在回收器完成年老年代对象的处理以及跨代引用的识别之后, 它将对年轻代对象进行追踪 (markNursery), 使用 incrementNew 方法对其进行标记、使用 sweepNursery 方法进行清扫,

并最终回收掉所有未被标记的对象。

年老代对象可以使用与延迟引用计数相同的策略进行回收。未被标记的、引用计数为零的、并非从根直接可达的对象都将得到回收（使用 `sweepZCT`）。如果某一引用计数为零的对象的脏标记已被设置，则回收器需要根据其在日志中的条目来对其（曾经的）子节点递归执行引用计数减少操作（如图 18.3 所示）；否则便意味着当前对象依然在使用中。

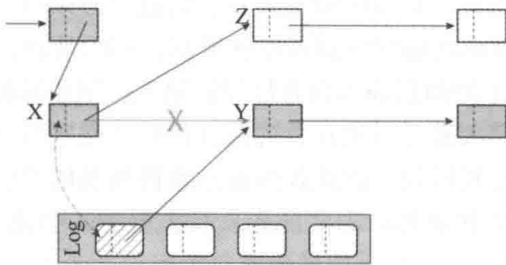


图 18.3 滑动视图允许回收器使用日志中的数据来对某一时刻堆的快照进行追踪。图中带阴影的对象表示在赋值器将从 X 到 Y 的引用改写为从 X 到 Z 的引用时对象图的状态。回收器可以通过对日志中 X 对应数据的追踪来获取对象图以前的状态

18.5.3 基于滑动视图的环状垃圾回收

到目前为止，我们所介绍的基于年龄的回收器均只能处理年轻代的环状垃圾，对于年老代中的环状垃圾则无能为力。Paz 等 [2007] 将 `Recycler` 的环状垃圾回收算法集成到基于年龄的回收器中。异步 `Recycler` 回收器所面临的主要难题在于其所处理的堆拓扑结构可能会发生变化，而滑动视图则能够（通过未修改对象的当前版本，或者已修改对象在日志中的副本）为回收器提供堆的一个固定快照（如图 18.3 所示），因此试验删除算法便可对其第一次所追踪的子图再次进行追踪。基于堆的滑动视图，我们便可使用简单的同步试验删除算法来处理环状垃圾，而不必使用 Bacon 和 Rajan[2001] 更加复杂的多颜色异步算法。

Paz 等提出了多种可以进一步减少试验删除所需遍历对象数量的优化策略。与 Bacon 和 Rajan[2001] 类似，他们也忽略了纯值对象等不可能构成环状垃圾的对象。只有经历多个回收周期之后还依然存活的成熟对象才会被当作备选垃圾，这便需要引入一个备选垃圾缓冲区队列，而不能只有一个备选垃圾缓冲区（他们发现，将备选垃圾的判定延迟两个回收周期最高效）。Paz 等同时还会避免将可能存活的对象作为备选垃圾，包括直接被根引用的对象、被窥探对象、在滑动视图收集完毕后又得到修改的对象。他们还额外引入一个 `markBlack` 阶段来对这些对象进行预处理，回收器会在该阶段将这些对象及其在滑动视图中的子节点着为黑色。但这一策略却存在一个问题：在回收过程中，存活性可以确定的对象集合（实际上是脏对象集合的一个子集）通常并不固定，因而我们不可能预测出在该对象变脏之前回收器将对其引用计数修改多少次，因此也不可能恢复其原有的引用计数值。所以说，试验删除算法只能基于另一个专门的环状引用计数进行操作。反之，如果回收器依然将这些对象当做备选垃圾，则引用计数回收器将不得不处理更多对象。

18.5.4 内存一致性

到目前为止，我们所介绍的滑动视图算法均假定处理器满足顺序一致性要求，但现代处理器通常达不到这一条件。对于赋值器而言，写操作应当确保：①写入日志的值必须是正确

370
372

的（也就是说，由于修改操作发生在回收开始之前，所以日志中的数据必须能够反应对象在被修改之前的状态）；②回收器在日志中读取到的数据必须是完整的；③在回收过程中，写屏障必须对即将写入对象域的引用进行窥探。在内存一致性较弱平台上，算法所引入的 4 次握手过程在一定程度上可以解决各操作之间的依赖问题（即确保了回收器在日志中读取到的数据是完整的，回收过程中的新引用都是经过窥探的），除此之外，算法还需要引入两个必要的改进：第一，对于赋值器而言，对写屏障写入日志的过程必须引入适当的同步机制，只有这样才能确保在真正更新指针域之前回收器可以读取到正确的日志，Levanoni 和 Petrank [2006] 所使用的方案是在 `log(src)` 的前后插入内存屏障；第二，回收器的某些操作也可能需要引入类似的同步机制，但是由于大部分对象在日志指针得到清空之后都不太可能再次被赋值器修改，所以这一策略可能会比较低效。因此在内存一致性较弱的平台上，回收器可以使用另一种策略来降低同步开销，即在处理日志之前先从日志缓冲区中读取一批数据到本地数组中。

[373]

18.6 需要考虑的问题

在并发环境下，引用计数算法所面临的首要问题是如何正确维护对象的引用计数。最简单的策略是要求赋值器在修改对象之前对其加锁，但如果锁相关操作的开销过大，则必须使用其他替代方案。并发解决方案的出发点在于避免赋值器线程之间为确保引用计数的一致性而引入的竞争。需要注意的是，内存管理器所关心的只是如何维护堆的一致性，至于赋值器线程之间的竞争是否会影响用户程序的正常执行，不是内存管理器应当关心的任务。

为确保一致性，我们必须考虑如何将指针写操作以及引用计数变更操作序列化。一种并不完美的解决方案是延迟引用计数，该策略会将垃圾对象的回收延迟，同时会将回收任务交给单独的回收线程来处理。但是，该方案却只能消除栈和寄存器中指针加载、写入时的引用计数操作开销，将指针写入堆中对象时依然需要立即执行引用计数变更操作。因此我们只能考虑如何将更新指针域时所必要的引用计数变更操作从赋值器线程转移到单个回收器线程。一种解决方案是每个赋值器线程将其引用计数变更操作缓冲到日志中，并周期性地将日志传递给回收器。合并引用计数对这一思想进行了拓展，该策略会记录对象在得到修改之前的快照，从而可以减少许多冗余的引用计数变更操作。这两种策略都将引用计数的变更与对象的回收从指针写操作中独立出来，并交由单独的回收器线程进行处理（当然也可以使用并行回收器线程）。如果能够获取堆在某一时刻的快照，也可简化并发环状引用计数算法。试验删除算法需要对某一子图进行多次遍历，通过遍历快照的方式，即使在赋值器并发修改对象图的情况下，回收器仍可以确保每次所追踪的将是相同的子图。

最后我们需要指出的是，大量文献都涉及如何在使用动态内存结构时安全地进行内存回收，最早可以追溯到 IBM 370 系统所使用的防 ABA 标签（ABA-prevention tag）。其他需要引入多字（multi-word）原子操作原语的无锁引用计数方案包括 Michael 和 Scott [1995]、Herlihy 等 [2002] 的方案。还有一些解决方案使用时间戳来将对象的回收延迟到可以安全执行这一操作时，但这一策略依赖于调度器，且易于导致单个线程出现延迟，甚至失败。例如，Linux 内核中所使用的读-复制-更新（Read-Copy-Update）策略 [McKenney 和 Slingwine, 1998] 会将对象的回收延迟到所有曾经访问过该对象的线程都达到某个“休眠”点（quiescence point）的时刻。其他使用立即引用计数（而非延迟引用计数）的策略通常需要特殊的编程模式，例如冒险指针（hazard pointer）[Michael, 2004] 或者通知（announcement）策略 [Sundell, 2005]。

[374]

实时垃圾回收

在前面的几章中我们介绍了并发回收算法与增量回收算法，它们的目的都是减少赋值器可以感知到的回收停顿时间。在并发回收中，回收器与赋值器在不同的处理器上同时工作，而在增量回收中，回收器与赋值器在相同的处理器上交替执行，且回收器每次仅处理很少一部分的回收工作。许多并发 / 增量回收算法都是针对那些对延迟十分敏感的应用而设计的，在这些应用程序中，较长的停顿时间可能会导致服务质量的下降（例如在图形用户界面中鼠标呈跳跃式移动）。因此，早期的并发回收器和增量回收器通常也称为实时（real-time）回收器，但是这些回收器仅能在特定条件下（例如限制对象的大小）满足实时系统的要求。在实时系统的定义得到明确之后，之前的各种回收算法都无法确保其能够支持真正的实时行为：它们均不能为赋值器提供较强的前进保障。如果赋值器（在读写屏障或者分配过程中）的某些操作必须加锁，则其前进保障必然受到影响。更加糟糕的是，在支持抢占式线程调度的系统中，并发执行的回收器线程可能会武断地中止赋值器线程的执行，进一步削弱了赋值器线程的前进保障。真正的实时回收（real-time collection, RTGC）必须能够精确地控制由垃圾回收所导致的赋值器中断，同时也必须确保系统不会在空间上超限。幸运的是，实时垃圾回收算法目前已经取得不少进展，从而将自动内存管理的适用范围拓展到实时系统领域。

19.1 实时系统

实时系统对应用程序中的某些特殊任务存在完成时间上的限制，即这些实时任务必须要在给定的时间窗内对系统输入（事件）进行响应。如果实时任务无法在给定时间内完成，要么会导致系统服务质量的下降（例如在数字视频的播放过程中出现丢帧），要么可能导致灾难性的系统失败（例如在错误的时间产生火花塞点火信号，从而对内燃机造成破坏）。因此，实时系统不仅必须满足逻辑上的正确性，其对实时事件的响应还必须满足时效性要求。

软实时系统（soft real-time system）能够容忍响应时间超限，但代价是服务质量下降，例如视频播放系统。过多的响应时间超限将导致服务质量无法接受，但偶尔出现响应时间超限并无大碍。Printezis[2006] 建议在设计针对软实时系统的垃圾回收器时，应当首先确定最大垃圾回收时间、系统时间片（time slice）的长度、可以接受的失败率。在给定时间片的任意时段内，垃圾回收器所占用的时间都不应当超过最大回收时间，且系统违背这一要求的次数必须控制在可以接受的失败率范围内。

比软实时系统要求更为严格的是硬实时系统（hard real-time system），此类系统一旦出现响应时间超限，则意味着严重的系统失败（例如在工程控制领域）。正确的硬实时系统必须确保所有实时约束条件都得到满足。面对这一时间上的约束条件，很有必要对实时系统中垃圾回收的响应能力从两方面进行描述：一方面反映出应用程序自身的要求（软 / 硬实时系统），另一方面则反映出垃圾回收器的行为 [Printezis, 2006]。

对于实时系统而言，程序执行的可预测性（predictability）比性能或者吞吐量更加重要。实时任务在时间上的行为表现应当在系统设计时便可预先确定，或者在测试过程中依

照经验来确定，因此其在运行过程中的响应时间便可提前预知（在一定的可信度范围内）。任务的最差执行时间（worst-case execution time, WCET）是指其在特定的硬件平台上独占式（即忽略重新调度）执行时所需的最大时间。多任务（multitask）实时系统在对任务进行调度时必须确保每个任务的时限要求均得到满足。要确保这些限制条件能够在运行时得到满足，设计者就必须事先基于特定的运行时调度算法（通常是基于优先级）进行可调度性分析（schedulability analysis）。

实时应用程序通常运行在针对特定目的而设计的嵌入式系统中，例如上文曾经提到的工程控制系统。单芯片处理器在嵌入式系统中占主导地位，因而增量垃圾回收技术可以很自然地迁移到嵌入式系统中，但随着多核嵌入式处理器变得越来越普遍，并发回收与并行回收在嵌入式领域中也开始占有一席之地。另外，与通用平台相比，嵌入式系统在空间方面的限制通常更加苛刻。

综上所述，万物静止式回收、并行回收，甚至是并发回收均会给赋值器引入不可预测的时间停顿，因此它们均不能满足实时系统的要求。传统垃圾回收器的回收工作量决于应用程序已用内存的总量与对象的大小、对象之间的内在联系、为满足未来分配需求所要达到的回收力度，因此回收器的调度情况可能会如图 19.1 所示。这一情况下，赋值器的处理器使用率不仅无法预测，也不能长期保持在同一水平。



图 19.1 传统回收器所产生的停顿在频度和持续时间上均无法预测。图中的灰色区域代表垃圾回收所导致的停顿

19.2 实时回收的调度

何时触发垃圾回收以及如何触发，是回收器影响赋值器执行的主要方面。对于万物静止式的回收器，只有当赋值器在内存分配的过程中感知到内存耗尽时才会唤起垃圾回收；增量回收器会要求赋值器在每次堆访问过程（通过读/写屏障）以及内存分配过程中承担一定的回收工作；并发回收器会在赋值器工作的同时并发执行回收工作，但仍需要引入赋值器屏障来确保回收器与赋值器之间的同步。为了维持稳定的空间消耗，回收器释放并回收死亡对象的速率应当能够匹配赋值器创建新对象的速率。内存碎片会造成空间上的浪费，进而有可能导致最差情况下赋值器的分配请求无法得到满足，除非回收器本身具有迁移存活对象的能力，或者可以借助于一个独立的整理过程。但是，迁移对象可能会引入额外的负担，从而可能影响到实时任务的完成。

[376]

到目前为止，研究者们已经提出了多种不同的实时垃圾回收调度策略，并且描述了不同策略下回收工作对赋值器的影响 [Henrikson, 1998 ; Detlefs, 2004b ; Cheng and Blelloch, 2001 ; Pizlo and Vitek, 2008]。基于工作的调度（work-based scheduling）策略会将回收工作分摊到赋值器的每个工作单元中。基于间隙的调度（slack-based scheduling）策略则会在实时任务的调度间隙执行回收工作（即当没有实时任务正在运行时）。如果实时任务的发生频率不高，或者只是周期性地发生（即简单地按照某一固有频率发生），则实时任务的调度间隙可能会在整体执行时间中占据相当大的比例。在支持优先级调度策略的系统中，令回收线程的优先级低于实时任务，便可简单地实现基于间隙的调度。基于时间的调度（time-based scheduling）策略会为回收器保留预定的独占式运行时间，在此期间赋值器线程将处于挂起状态，该策略可以确保系统能够满足预定的最小赋值器使用率要求。

19.3 基于工作的实时回收

经典的 Baker[1978] 增量式半区复制回收器是最早在实时垃圾回收领域进行尝试的回收算法之一。该算法基于对象（此处是指 Lisp 语言的 cons 单元）大小固定这一限制提出了一种分析实时行为的精确模型。我们曾在第 15 章介绍过，Baker 式读屏障会把赋值器即将访问的来源空间对象复制到目标空间，从而阻止赋值器访问来源空间中的对象。由于对象的大小都是固定的，所以单个读屏障的工作量是有界的。与此同时，赋值器在每次内存分配过程中也会执行有限的回收工作（即扫描固定数量的灰色目标空间域，并在必要时复制其大小固定的来源空间目标）。分配过程扫描的域越多，则回收过程结束得越快，但赋值器执行得也更慢。Baker[1978] 给出了其回收器在时间和空间方面的界限：其空间界限为 $2R(1 + 1/k)$ ，其中 R 为可达空间大小， k 为可调整的时间界限，它是由内存分配过程中赋值器所需扫描的域的数量决定的。Baker 并未给出变长数组的增量式复制方法，但这一问题并非其所关注的主要方面。

19.3.1 并行、并发副本回收

Blelloch 和 Cheng [1999] 将 Baker[1978] 的分析拓展到多处理器环境，他们提出了一种并发、并行副本复制回收算法，并推导出了该回收器在时间和空间上的界限。在后续实现该回收器的过程中，他们最早提出用最小赋值器使用率（minimum mutator utilisation）这一概念来描述回收器给赋值器所带来的干扰 [Cheng and Blelloch, 2001]。由于他们的回收器依然是基于工作的，所以不论该回收器付出多大的努力来最大限度地降低停顿时间，其停顿时间的分布仍会出现不可预测的变化，从而导致赋值器在满足实时要求方面存在一定的困难。我们将在 19.5 节看到，最小赋值器使用率也可以用于引导基于时间的实时垃圾回收调度，此时最小赋值器使用率将成为回收器的一个输入约束条件。另外，对于如何在严格满足时间界限要求的同时满足空间界限要求，Blelloch 和 Cheng 为后来者提供了十分有用的启示，接下来我们将介绍其具体设计实现。

机器模型 (machine model)。Blelloch 和 Cheng 提出了一种理想化的机器模型，而真正的回收器实现则必须尽力弥补真正的机器模型与这一理想化模型之间的差异。该模型假定机器本身是一个典型的对称共享内存多处理器，且提供了可以用于同步操作的 TestAndSet 和 FetchAndAdd 指令。这些原语可以直接由硬件支持，也可使用 LoadLinked/StoreConditionally 或者 CompareAndSwap 原语在现代对称多处理器上简单地实现，但其同时要求 FetchAndAdd 指令必须以公平的方式实现，从而确保每个处理器的前进保障。他们还假定可以使用一种简单的中断方式来实现每个处理器上增量回收的开始与结束，这可以通过我们在 11.6 节所描述的安全回收点来实现。更加重要的是，他们假定内存访问模式满足顺序一致性，这一要求给回收器的具体实现带来了更高的挑战，因为某些内存访问指令必须以适当的方式排序以确保正确性。

该模型中的内存空间可以看作是一组连续位置的集合，其地址范围是 $[2..M + 1]$ （值为 0 或者 1 的指针具有特殊含义），其中 M 是系统的最大内存大小。每个位置都至少可以持有一个指针。

为简化时间分析，他们将系统执行一条指令所需花费的最长时间当作是每条指令的执行时间（指令之间的中断不计入该时间）。

应用模型 (application model)。应用模型假定赋值器基于传统的 Read、Write 以及

$New(n)$ 操作进行工作, 其中后者会分配一个包含 n 个域的对象并返回其首个域的地址, 同时该对象还包括一个供内存管理器使用的头部字。除此之外, Blleloch 和 Cheng 还假定每个处理器在调用 $New(n)$ 之后都会立即调用 n 次 $InitSlot(v)$ 来对新创建对象的 n 个域进行初始化, 其中 n 从 0 开始。处理器在使用该对象之前, 或者调用下一次 New 操作之前, 必须确保这 n 次 $InitSlot$ 调用已经完成, 而 $Read$ 和 $Write$ 操作则可以与 $InitSlot$ 操作以任意方式穿插调用。另外, 理想化的应用模型假定 $Write$ 操作是原子化的 (即任意两个处理器的 $Write$ 操作都不可能发生重叠)。内存管理器同时还支持 $isPointer(p, i)$ 函数来判定指针 p 所引用对象的第 i 个域是否为指针, 其返回结果通常是由对象的静态类型决定的, 而在面向对象语言中则是由其所属的类决定的。

算法实现 (the algorithm)。回收器大体上以 Nettles 和 O'Toole[1993] 的副本复制回收器作为原型进行设计, 但其不同之处在于, 赋值器所遵守的并非来源空间不变式, 也并不需要对赋值器更新进行记录, 而是遵守副本不变式 (replication invariant): 在回收过程中, 如果赋值器需要更新某一对象, 则其必须同时更新对象主体及其副本 (如果存在的话); 在回收过程中, 当分配器分配新对象时, 其不仅要在来源空间中创建出对象主体, 也要在目标空间中创建出它的副本。Blleloch 和 Cheng 还使用 Yuasa[1990] 式起始快照删除写屏障来确保算法的正确性。

Blleloch 和 Cheng 假定每个对象主体 p 都存在一个头域 $forwardingAddress(p)$, 与此同时, 每个对象副本 r 都存在一个头域 $copyCount(r)$ (这两个域可以复用相同的槽, 因为两个域不可能同时出现在同一副本中)。该头域在回收过程中可以扮演多个角色: 在对象主体上进行同步, 以便控制由哪个线程生成副本、用作指向对象副本的转发指针; 在对象副本中记录还有多少域需要从主体复制到其中、在赋值器以及复制线程操作对象副本时确保同步。当对象主体 p 为白色时, 将只存在主体而没有副本, 因而其头域的值为零 (即 $forwardingAddress(p) = null$); 如果对象变为灰色, 且回收器已完成其副本 r 的空间分配, 则对象主体中头域的值将变成指向副本的转发指针 (即 $forwardingAddress(p) = r$), 而副本 r 中头域的值则为待复制域的数量 (即 $copyCount(r) = n$); 当对象变为黑色时 (即复制完成), 其副本头域的值将为零 ($copyCount(r) = 0$)。

算法依照图 19.2 所示的方式将堆组织为两个半区。 $fromBot$ 和 $fromTop$ 两个变量组成了来源空间的边界, 且它们都属于线程私有变量。回收器在目标空间的顶部显式维护一个复制栈用以持有灰色对象的引用。我们在 14.6 节曾经提到, Blleloch 和 Cheng 声称, 对于并发回收线程之间在共享复制工作时的局部性以及同步问题, 显式复制栈会比 Cheney 队列更加容易控制。所有的副本以及新分配对象均位于变量 $toBot$ 和 $free$ 之间的区域, 而复制栈则位于变量 $sharedStack$ 和 $toTop$ 之间的区域 (栈的增长方向为从 $toTop$ 到 $sharedStack$)。如果 $free = sharedStack$, 则意味着内存耗尽, 此时如果回收器没有启动, 其将被激活, 否则系统将抛出内存错误。变量 $toBot$ 和 $toTop$ 也为线程私有变量而 $free$ 和 $sharedStack$ 则为共享变量。

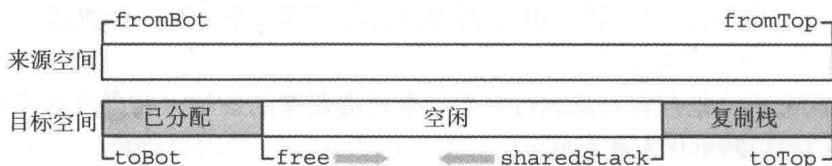


图 19.2 Blleloch 和 Cheng 基于工作的回收器: 堆结构

在算法 19.1 中, `copyOnSlot` 方法展示了将对象主体中的某个槽复制到其副本的过程, 该方法以待复制灰色对象主体 `p` 作为参数, 并依照副本中所记录的待复制域数量进行复制, 同时将其所复制域的目标对象着为灰色 (调用 `makeGrey` 方法), 最后再减少副本中待复制域的数量。函数执行完毕后, 如果 `p` 中依然存在待复制域, 则其将依然保持灰色, 回收器会再次将其压入本地复制栈中 (针对本地栈的操作可以参见算法 14.8)。

算法 19.1 Blelloch 和 Cheng 基于工作的回收器: 复制算法

```

1  shared gcOn ← false
2  shared free                                     /* 分配基址 */
3  shared sharedStack                             /* 复制栈的栈顶指针 */
4
5  copyOnSlot(p):                                /* p 为灰色主体对象 */
6      r ← forwardingAddress(p)                   /* 指向副本的指针 */
7      i ← copyCount(r)-1                         /* 待复制槽的索引值 */
8      copyCount(r) ← -(i+1)                     /* 加锁, 以避免复制过程中赋值器进行写操作 */
9      v ← p[i]
10     if isPointer(p, i)
11         v ← makeGrey(v)                       /* 如果 v 是指针, 则将其着为灰色 */
12     r[i] ← v                                   /* 复制该槽 */
13     copyCount(r) ← i                           /* 将该对象解锁, 同时将待复制索引号递减 */
14     if i > 0
15         localPush(p)                          /* 重新压入本地复制栈 */
16
17  makeGrey(p):                                  /* p 必须为主体对象 */
18     if TestAndSet(&forwardingAddress(p)) ≠ 0 /* 通过竞争的方式来获取转发地址写入权 */
19         /* 竞争失败 */
20         while forwardingAddress(p) = 1
21             /* 等待: 直到转发地址变为合法地址 */
22     else
23         /* 竞争胜出 */
24         count ← length(p)                       /* 对象主体的长度 */
25         r ← allocate(count)                     /* 分配副本 */
26         copyCount(r) ← count                    /* 在副本中设置待复制域的数量 */
27         forwardingAddress(p) ← r                 /* 为主体对象设置转发地址 */
28         localPush(p)                            /* 将主体对象压入复制栈 */
29     return forwardingAddress(p)
30
31  allocate(n):
32      ref ← FetchAndAdd(&free, n)
33      if ref + n > sharedStack                    /* 目标空间是否耗尽? */
34          if gcOn
35              error "Out of memory"
36              interrupt(collectorOn)              /* 中断赋值器, 并启动下一轮回收 */
37              allocate(n)                         /* 重试 */
38      return ref

```

`makeGrey` 函数会将白色对象 (即尚未创建副本的对象) 着为灰色, 并返回其副本的地址。由于多处理器可能同时尝试将某一对象着色, 所以算法使用 `TestAndSet` 原语来检测对象是否依然为白色, 从而避免了同一对象出现多个副本的情况出现。我们将从“复制-复制”竞争中胜出的处理器称为“指定复制者” (designated copier)。`makeGrey` 函数需要对 `forwardingAddress(p)` 可能出现的 3 种情况进行区别对待:

1) `TestAndSet` 操作返回零, 表明当前处理器成为“指定复制者”, 然后该处理器将会: 在目标空间中为副本 `r` 分配空间、将其头域 `copyCount(r)` 设置为对象的长度、将对象主体

的头域 `forwardingAddress(p)` 设置为副本 `r` 的地址、将对象主体的引用压入私有复制栈中，最后返回副本 `r` 的引用。

2) `TestAndSet` 操作返回非零，且头域的值为合法的转发地址，此时 `makeGrey` 方法只需简单地返回副本的引用。

3) `TestAndSet` 操作返回非零，但其头域的值为 1，这意味着“指定复制者”正在为该对象创建副本，但尚未设置转发指针。此时，当前处理器必须等待到该头域变成合法的转发地址为止。

算法 19.2 展示了回收过程中赋值器各项操作的执行代码。`New` 操作使用 `allocate` 方法来为对象主体及其副本分配空间，同时为后续即将调用的 `InitSlot` 设置必要的参数，以便其完成对象的初值设定：变量 `lastA` 记录了上一个新分配对象的地址，`lastL` 记录了其长度，而 `lastC` 记录了其中有多少个槽已经完成初始化。`InitSlot` 函数将使用下一个槽的初值同时初始化对象主体及其副本中的对应槽，然后增加 `lastC`。为了维持“禁止黑色对象持有白色引用”的强三色不变式要求，`InitSlot` 方法需要将所有指针着色。赋值器每分配一个字，`collect(k)` 方法便会增量式的复制 k 个字。从设计上讲，算法允许回收器在某一对象部分初始化的情况下（即某一处理器的 `lastC \neq lastL`）开始新一轮回收。

算法 19.2 Blleloch 和 Cheng 基于工作的回收器：赋值器操作 (`gcOn=true`)

```

1  lastA                                /* 每处理器指针变量，指向上一个新分配的对象 */
2  lastL                                /* 每处理器变量，代表上一个新分配对象的长度 */
3  lastC                                /* 每处理器变量，记录上一个新分配对象中有多少个槽已完成初始化 */
4
5  Read(p, i):
6      return p[i]
7
8  New(n):
9      p ← allocate(n)                  /* 分配对象主体 */
10     r ← allocate(n)                  /* 分配对象副本 */
11     forwardingAddress(p) ← r /* 将对象主体的转发地址设置为其副本的地址 */
12     copyCount(r) ← 0                 /* 已完成复制的槽数量为零 */
13     lastA ← p                        /* 设置上一个新分配对象的指针 */
14     lastC ← 0                        /* 设置已初始化槽的数量 */
15     lastL ← n                        /* 设置对象长度 */
16     return p
17
18  atomic Write(p, i, v):
19      if isPointer(p, i)
20          makeGrey(p[i])              /* 将被删除的引用着色 */
21      p[i] ← v                        /* 将新值写入对象主体 */
22      if forwardingAddress(p) ≠ 0      /* 判断对象是否已被转发 */
23          while forwardingAddress(p) = 1
24              /* 等待头域设置为正确的转发地址 */
25              r ← forwardingAddress(p) /* 获取副本引用 */
26              while copyCount(r) = -(i+1)
27                  /* 等待复制线程完成目标槽的复制 */
28                  if isPointer(p, i)
29                      v ← makeGrey(v) /* 使用已完成着色的新引用来更新副本的对应槽 */
30                      r[i] ← v        /* 更新副本 */
31              collect(k)               /* 执行 k 个复制增量 */
32
33  InitSlot(v):                        /* 初始化上一个新分配对象的下一个槽 */
34      lastA[lastC] ← v                /* 初始化对象主体 */
35      if isPointer(lastA, lastC)

```

```

36     v ← makeGrey(v) /* 使用已完成着色的新引用来初始化副本中对应的槽 */
37     forwardingAddress(lastA)[lastC++] ← v          /* 初始化副本 */
38     collect(k)                                     /* 执行 k 个复制增量 */

```

Write 操作首先把将被覆盖（删除）的指针着为灰色（以保持快照可达性），然后再将新值写入对象主体及其副本（如果存在的话）中的对应域。当赋值器针对某一灰色对象执行写操作时，“指定复制者”可能正在复制相同的槽，这一“复制-写入”竞争有可能导致赋值器更新的丢失，即：如果赋值器更新副本中对应槽的操作发生在复制线程从对象主体读取数据之后、将其写入副本之前，则赋值器在副本中的更新操作将被覆盖。因此，Write 操作不仅要等待“指定复制者”完成对象副本的创建，还要等待其完成目标槽的复制。但是，如果 Write 操作在“指定复制者”对目标槽加锁之前执行写入，并不会出现问题，此时“指定复制者”最多只是重复写入赋值器所写入的值。在 Write 操作中，两处可能导致赋值器等待的 while 语句均存在时间上界，第一处的等待时间上界是由“指定复制者”分配副本的时间决定的，而第二处的等待时间上界则取决于“指定复制者”复制一个槽的时间。

[380]

算法之所以使用 InitSlot 操作而非 Write 操作来设置对象初值，原因是前者的开销更小：首先，未初始化槽的值天然为 null，因而无需为其引入删除屏障来维护快照的完整性；其次，新分配对象通常都会存在副本，因而无需额外检测副本是否存在；最后，算法允许回收器在某一对象尚未完全初始化的情况下启动新一轮回收，此时回收器只需复制已经初始化的槽（见算法 19.4 中的 collectorOn 方法）。

[381]

算法 19.3 展示了扮演回收器角色的函数 collect(k)，该函数会实现 k 个槽的复制。共享复制栈允许处理器之间分摊复制任务。为减少潜在开销较大的 sharedPop 操作的调用次数（该函数需要依赖 FetchAndAdd 原语）、提升本地优化几率、提升回收工作的局部性，每个处理器所处理的工作大都是从其本地复制栈中获取的（共享栈与本地栈的相关操作参见算法 14.8）。只有当本地栈为空时，处理器才会从共享栈中获取更多的工作。复制完 k 个槽之后，collect 方法会将剩余工作归还共享栈。需要注意的是，在任意时刻，第 2 ~ 10 行复制槽的工作都不可能与第 10 ~ 12 行将剩余工作归还共享栈的工作同时发生，这是由算法 14.9 所介绍的“工作空间”机制所保证的，详见 14.6 节。

算法 19.3 Bbleloch 和 Cheng 基于工作的回收器：回收器代码

```

1  collect(k):
2      enterRoom()
3      for i ← 0 to k-1
4          if isLocalStackEmpty() /* 本地栈为空 */
5              sharedPop() /* 从共享栈中获取部分工作，并压入本地栈 */
6              if isLocalStackEmpty() /* 本地栈依然为空 */
7                  break /* 无更多回收工作 */
8              p ← localPop()
9              copyOneSlot(p)
10             transitionRooms()
11             sharedPush() /* 将剩余工作迁移到共享栈 */
12             if exitRoom()
13                 interrupt(collectorOff) /* 关闭回收器 */

```

算法 19.4 展示了回收器的启动（collectorOn）与停止（collectorOff）代码。此处，我们假定所有的根都位于每个处理器固定数量的寄存器 REG 中。synch 方法所扮演的是同步屏

障的角色，它可以确保所有处理器都在该屏障汇聚之后再往下执行，其目的在于确保每个处理器能够针对 `gcOn`、`free`、`sharedStack` 这三个变量的值产生一致的判定。新一轮回收周期开始后，每个处理器首先会为最后一个新分配对象创建副本，并将副本中头域的值初始化为 `lastC`，这意味着最后一个新分配对象中只有已完成初始化的域才需复制到副本中。在回收周期结束后，寄存器以及 `lastA` 都将更新为其目标对象的副本地址。

算法 19.4 Blelloch 和 Cheng 基于工作的回收器：回收器的启动与停止

```

1  shared gcOn
2  shared toTop
3  shared free
4  shared count  $\leftarrow$  0                /* 已完成同步的处理器数量 */
5  shared round  $\leftarrow$  0              /* 当前同步轮数 */
6
7  synch():
8      curRound  $\leftarrow$  round
9      self  $\leftarrow$  FetchAndAdd(&cnt, 1) + 1
10     if self = numProc    /* 本轮同步完成，重置数据，以便启动下一次回收 */
11         cnt  $\leftarrow$  0
12         round++
13     while round = curRound
14         /* 等待，直到最后一个处理器修改 round 为止 */
15
16 collectorOn():
17     synch()
18     gcOn  $\leftarrow$  true
19     toBot, fromBot  $\leftarrow$  fromBot, toBot    /* 来源空间、目标空间翻转 */
20     toTop, fromTop  $\leftarrow$  fromTop, toTop
21     free, sharedStack  $\leftarrow$  toBot, toTop
22     stackLimit  $\leftarrow$  sharedStack
23     synch()
24     r  $\leftarrow$  allocate(lastL)                /* 为最后一个新分配对象创建副本 */
25     forwardingAddress(lastA)  $\leftarrow$  r    /* 为最后一个新分配对象设置转发地址 */
26     copyCount(r)  $\leftarrow$  lastC            /* 设置待复制槽的数量 */
27     if lastC > 0
28         localPush(lastA)                /* 将剩余的复制工作压入本地复制栈 */
29         for i  $\leftarrow$  0 to length(REG)    /* 将根着为灰色 */
30             if isPointer(REG, i)
31                 makeGrey(REG[i])
32         sharedPush()                    /* 将本地复制栈中的对象迁移到共享栈 */
33         synch()
34
35 collectorOff():
36     synch()
37     for i  $\leftarrow$  0 to length(REG)        /* 将根着为灰色 */
38         if isPointer(REG, i)
39             REG[i]  $\leftarrow$  forwardingAddress(REG[i])    /* 对根进行转发 */
40     lastA  $\leftarrow$  forwardingAddress(lastA)
41     gcOn  $\leftarrow$  false
42     synch()

```

其他具有实际意义的改进。实时副本复制算法的原始版本 [Blelloch and Cheng, 1999] 及其后续改进版本 [Cheng and Blelloch, 2001; Cheng, 2001] 都针对上述算法做出了不少具有实际意义的改进：为避免在每次 `allocate` 过程中（第 32 行）调用 `FetchAndAdd` 操作，每个处理器可以使用 7.7 节所描述的本地分配缓冲区策略进行分配；在 `makeGrey` 方法中，由

于处理器必然知道其本地分配缓冲区中下一个将要分配的对象地址，所以算法可以使用 CompareAndSwap 操作来替代 TestAndSet，从而避免了单纯的自旋操作。其他改进措施包括：将 New 和 InitLoc 方法中的回收工作延迟到每个本地分配缓冲区都被（小）对象填满时，从而避免了 New 方法中两次内存分配的开销（即对象主体及其副本）；使用工作空间同步机制将 Write 操作原子化（即在任意时刻，只有一个处理器可以进入“写工作空间”）。

时间与空间上界。算法付出了相当大的努力来确保每个回收增量的工作存在上界，从而使得系统在垃圾回收方面的时间和空间开销存在精确的上界。Blleloch 和 Cheng[1999] 已经证明，算法所需空间的上界为 $2(R(1+2/k)+N+5PD)$ 个字，其中 P 为处理器数量， R 为一次计算的最大可达空间（即从根集合可达的字的数量）， N 为最大可达对象的数量， D 为对象的最大深度， k 为系统每分配出一个字的同时需要完成多少个字的复制，该参数可以对系统在时间和空间方面的开销进行平衡。他们同时指出，赋值器线程的最大停顿时间不会超过某一正比于 k 的值（以非阻塞机器指令数为单位）。由于 makeGrey 方法一次仅处理灰色波面中的一个域而非一个对象，所以即使对于大对象以及数组，该算法的时空上限也能得到保证。

性能。Blleloch 和 Cheng[2001] 在 ML（一种静态类型函数式语言）中实现了该回收器。ML 应用程序通常具有极高的内存分配率，这对大多数回收器而言这都是一项挑战。他们基于 Sun Enterprise 10000（包含 64 个处理器，每个处理器的时钟频率大约为数百兆赫兹）进行实验，结果表明，如果仅使用单处理器，该回收器的开销平均会比对等的万物静止式回收器高出 51%（基于多个基准程序测试得出），其中 39% 的开销源于对并行回收的支持，12% 的开销源于对并发回收的支持。当可用处理器数量扩大到 32 个时，回收器表现出较好的递增适应性（17.2 倍的加速），而赋值器的递增适应性则表现较差（9.2 倍的加速），整个系统在可用处理器数量为 8 时表现最佳，此时回收器和赋值器分别得到了 7.8 倍和 7.2 倍的加速。以 10ms 作为测量单位，所有万物静止式回收器的最小赋值器使用率均为零或者接近于零，相比之下，当 $k=2$ 时，该回收器的最小赋值器使用率可以达到 10%，而当 $k=1.2$ 时，可以达到 15%。该回收器的最大停顿时间大约为 3 ~ 4ms。

19.3.2 非均匀工作负载的影响

对于基于工作的实时垃圾回收调度策略，其所面临的最大问题是：赋值器与回收器操作之间的紧耦合导致整个系统的最小赋值器使用率并不均匀。基于工作的复制式回收器必须假定在最差情况下所有赋值器均在执行堆操作，此时对于 Baker[1978] 回收器而言，读取一个指针槽可能就需要复制其目标对象。对于 Lisp 的 cons 单元，其复制开销是固定的，而对于数组等变长对象，复制整个对象的开销便可能成为问题。分配过程同样会涉及固定量的回收工作。在回收周期开始时，回收器还需要对来源空间和目标空间进行翻转、扫描根并复制其目标对象，这通常会涉及全局变量（在特定程序中数量存在上界）以及局部（线程栈）变量（其上界为将线程栈填满）。总之，回收器可能遭遇的最差情况与一般情况的差别太大，以至于最差情况下的执行时间分析对实际情况下的可调度性分析并无指导意义。

研究者们提出了多种策略来遏制基于工作的回收调度策略在最差情况下的回收开销。为了限制栈扫描工作量，Cheng 和 Blleloch[2001] 将栈划分为固定大小的子栈（stacklet），回收器在翻转过程中只需要扫描活动赋值器线程最顶端的子栈，而其他子栈则将延迟到适当的时机进行扫描。为阻止赋值器线程返回到未经扫描的子栈中，该策略需要引入栈屏障来拦截赋

值器线程的退栈操作，并迫使赋值器线程在返回到未经扫描的子栈之前对其进行扫描。对于赋值器线程尝试返回到正在由回收器扫描的子栈这一情况，Detlefs[2004b]提出了两种解决方案：一是赋值器等待回收器工作完成，二是回收器中止对该子栈的扫描，并将扫描工作交由赋值器完成。

类似地，为限制回收器在推进回收波面时的扫描/复制工作粒度，可以将变长对象拆分为固定大小的子对象（oblet）、将数组拆分为子数组（arraylet）。当然，这些非标准的表现形式要求系统在访问对象时、对数组元素执行索引操作时做出相应的调整，这一额外的间接访问模式进一步增加了系统在时间和空间方面的开销 [Siebert, 1998, 2000, 2010]。

尽管研究者们提出了多种限制回收工作粒度的策略，但 Detlefs 依然认为，不均匀的工作负载将是压垮纯粹基于工作的调度策略的最后一根稻草。例如，在 Baker 式并发复制回收器中，赋值器的操作开销会在回收周期的不同时段存在显著差别：在翻转操作之前，赋值器仅需要在偶尔发生的分配过程中付出一些额外开销（目的是确保回收波面向前推进），此时读操作在大多数情况下都会访问已经完成复制的对象；而在回收器执行翻转且仅完成赋值器根的扫描之后，由于几乎每个读操作都需要复制其目标对象，所以读操作的平均开销可能会接近理论上的最大值。类似地，在 Blleloch 和 Cheng[1999] 的回收器中，尽管写操作的发生频率远小于读操作，但不同时段的操作需要为目标对象创建副本的概率也存在很大差别。

在工作负载不均衡的情况下，尽管基于工作的调度策略依然可以确保停顿时间的短暂性与可预测性，但由于回收工作的频率和持续时间依然可能降低赋值器的使用率。以图 19.3 为例，尽管回收器可以确保回收停顿时间不大于 1ms，但赋值器却只能利用两次回收之间 0.1ms 的间隙进行工作。这种情况下，即使回收器能够保证可预测的短时停顿，留给赋值器的执行时间也不足以满足实时系统的时限要求。

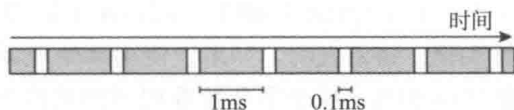


图 19.3 即使回收停顿时间很短，赋值器使用率依然可能很低。此时绝大多数 CPU 时间被回收器占用（图中的灰色部分），而赋值器只能在回收停顿的间隙（图中的白色部分）进行工作

基于工作的调度策略不仅可能导致赋值器操作的额外负载不均衡，而且平均负载与最差情况下的负载情况存在较大差异，这便导致对基于工作的调度策略进行最差执行时间的分析意义不大：为垃圾回收超量分配不必要的处理器资源必然会降低赋值器使用率。

在非复制式并发回收器中，赋值器写屏障仅需要简单地将来源对象或者新/旧目标对象着色，因而赋值器堆访问操作的开销相对较小且有界。但是，由于分配过程会爆发式地出现，所以基于工作的调度策略仍然可能导致赋值器的垃圾回收开销存在较大的不均衡。

因此，更加高级的调度策略并不会将回收工作完全当作赋值器的额外开销来对待，而是将其当作另一种必须完成的实时任务来进行调度。如此一来，最差情况下的赋值器执行时间分析才会与真正的赋值器平均性能接近，从而达到更好的处理器使用率。对于某些发生频率较低但开销较大的操作（例如翻转赋值器），回收器只需确保其执行时间足够短，且能够在回收器时间片内执行完毕即可。

385

19.4 基于间隙的实时回收

针对实时回收器的调度问题，Henriksson 认为，垃圾回收工作应当避开高优先级（实时）任务的执行 [Magnusson and Henriksson, 1995; Henriksson, 1998]，即只有当系统中不存在

等待执行的高优先级任务时，垃圾回收工作才可以执行。高优先级任务的内存分配操作不会承担任何回收相关开销，而低优先级任务的分配过程则需要承担一定的垃圾回收任务。系统中还存在一种特殊的任务，即高优先级垃圾回收（high-priority garbage collection）任务，其目的在于完成高优先级任务执行过程中所忽略的回收相关工作，例如应当由高优先级任务在分配过程中执行的回收工作。高优先级回收任务的优先级低于高优先级任务，但却高于低优先级任务。回收器必须确保系统中有足够的已初始化空闲内存，并且能够满足高优先级任务的内存分配要求。因此，回收相关工作完全是在实时任务的调度间隙中执行的。

回收器将堆布置为两个半区的形式，如图 19.4 所示。赋值器将从目标空间的顶端分配对象，即指针 `top` 所指向的位置；得到迁移的对象将被置于目标空间的底部，即指针 `bottom` 所指向的位置。回收器使用传统的 Cheney 扫描策略来完成所有已迁移对象的扫描，并将它们所引用的来源空间对象迁移到目标空间。当低优先级线程在目标空间顶部分配新对象时，其会增量式地执行一些迁移工作。指针 `scan` 能够反映出回收器扫描已迁移对象的进度。

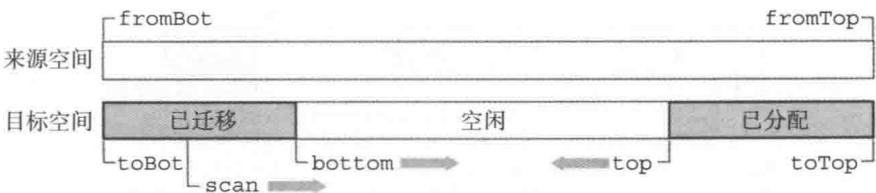


图 19.4 Henriksson 基于间隙的回收器：堆结构

Henriksson 的回收器使用 Brooks 间接屏障来拦截所有堆访问操作，同时还使用 Dijkstra 插入屏障来确保新目标对象必然位于目标空间（如果不在，则将其复制中）。该屏障可以确保回收器满足强三色不变式，即任何目标空间对象均不包含来源空间对象的引用。但是，Henriksson 并不要求高优先级任务在写屏障中复制完整的对象，而是采用懒惰迁移策略，即：写屏障仅需要简单地为目标空间副本分配空间，但不需要将来源空间对象的内部数据复制到其中。垃圾回收器终究会得到执行（可能是以高优先级回收任务的方式执行，也可能在低优先级任务的分配过程中得到执行），当其扫描到目标空间尚未填充的副本时会完成被延迟的复制工作。回收器在扫描目标空间副本之前，必须先将来源空间主体中的数据复制到其中。为避免赋值器访问尚未完成复制的空目标空间副本，Henriksson 对 Brooks 写屏障进行扩展，即在每个目标空间外壳中记录一个指向其在来源空间原始对象的反向指针。这一懒惰迁移策略如图 19.5 所示。

算法 19.5 与算法 19.6 描述了回收器的大体执行流程，整个回收器与算法 17.1 所描述的并发复制回收器十分类似，但该回收器额外使用的 Brooks 式间接屏障放宽了赋值器的目标空间不变式的要求，同时写屏障也会将复制对象内部数据的任务推迟给回收器完成（与 Sapphire 回收器类似）。需要注意的是，即使赋值器依然工作在来源空间，回收器仍可以借助于临时性的 `toAddress` 指针完成目标空间副本中引用的转发，因为 `toAddress` 指针所占用的指针域与 `forwardingAddress` 头域是两个不同的域。

回收器本身以协程（coroutine）方式实现，因此垃圾回收任务与低优先级赋值器任务可以通过代码中的 `yield` 点来实现交替执行，而高优先级任务在任意时刻都可以通过抢占（preempt）的方式重新获取程序的控制权。如果抢占行为发生在回收器复制某一对象的过程中，则复制过程会简单地中止（abort），并在回收器恢复执行时重新启动。另外，Henriksson

假定回收器是基于单处理器平台执行的，因而通过禁止调度器中断的方式便足以实现原子操作。

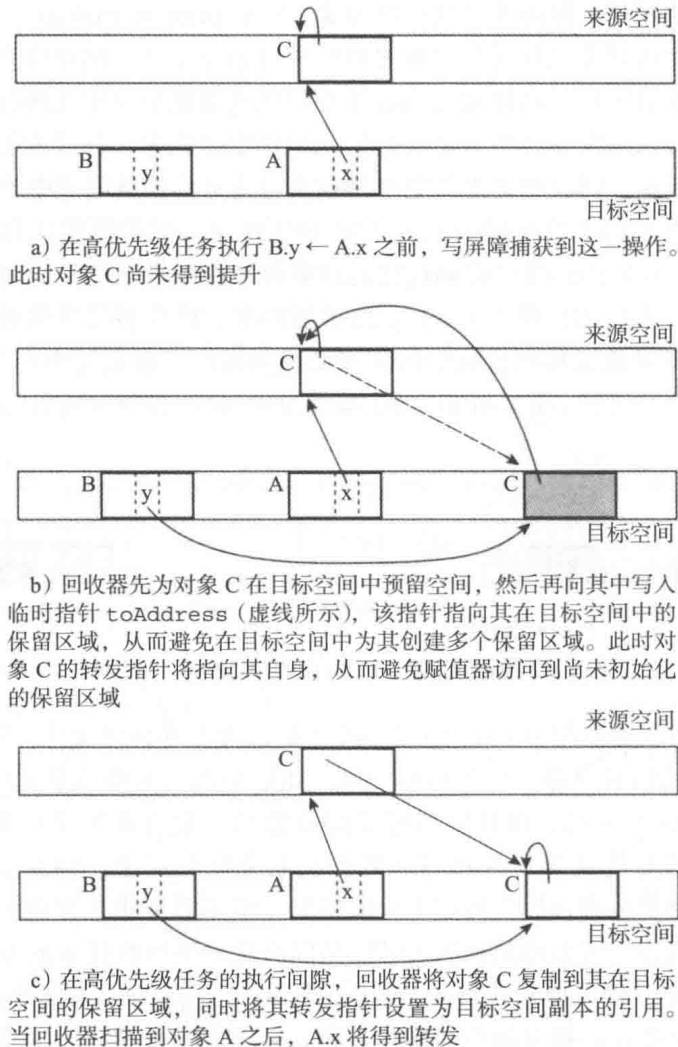


图 19.5 Henriksson 基于间隙的回收器：懒惰迁移

见 Henriksson [1998] 一书，经许可后转载

算法 19.5 Henriksson 基于间隙的回收器

```

1  coroutine collector:
2      loop
3          while bottom < top                                /* 目标空间尚未填满 */
4              yield                                         /* 转移到赋值器执行 */
5              flip()
6              for each fld in Roots
7                  process(fld)
8                  if not behind()
9                      yield                                  /* 转移到赋值器执行 */
10             while scan < bottom
11                 scan ← scanObject(scan)
12                 if not behind()
13                     yield                                  /* 转移到赋值器执行 */
14 
```

```

15 flip():
16     toBot, fromBot ← fromBot, toBot
17     toTop, fromTop ← fromTop, toTop
18     bottom, top ← toBot, toTop
19     scan ← bottom
20
21 scanObject(toRef):
22     fromRef ← forwardingAddress(toRef)
23     move(fromRef, toRef)
24     for each fld in Pointers(toRef)
25         process(fld)
26     forwardingAddress(fromRef) ← toRef
27     return toRef + size(toRef)
28
29 process(fld):
30     fromRef ← *fld
31     if fromRef ≠ null
32         *fld ← forward(fromRef)          /* 使用目标空间引用进行更新 */
33
34 forward(fromRef):
35     toRef ← forwardingAddress(fromRef)
36     if toRef = fromRef                    /* 尚未得到迁移 */
37         toRef ← toAddress(fromRef)
38     if toRef = null                      /* 并未得到迁移 (依然处于未标记状态) */
39         toRef ← schedule(fromRef)
40     return toRef
41
42 schedule(fromRef):
43     toRef ← bottom
44     bottom ← bottom + size(fromRef)
45     if bottom > top
46         error "Out of memory"
47     toAddress(fromRef) ← toRef          /* 为对象的迁移做调度准备 (进行标记) */
48     return toRef

```

算法 19.6 Henriksson 基于间隙的回收器：赋值器操作

```

1 atomic Read(src, i):
2     src ← forwardingAddress(src)          /* Brooks 间接屏障 */
3     return src[i]
4
5 atomic Write(src, i, ref):
6     src ← forwardingAddress(src)          /* Brooks 间接屏障 */
7     if ref in fromspace
8         ref ← forward(ref)
9     src[i] ← ref
10
11 atomic NewHighPriority(size):
12     top ← top - size
13     toRef ← top
14     forwardingAddress(toRef) ← toRef
15     return toRef
16
17 atomic NewLowPriority(size):
18     while behind()
19         yield                            /* 唤醒回收器 */
20     top ← top - size
21     toRef ← top
22     if bottom > top

```

```

23     error "Out of memory"
24     forwardingAddress(toRef) ← toRef
25     return toRef

```

19.4.1 回收工作的调度

回收器在每个回收增量中所需执行的工作量（通过 `behind` 方法进行控制）必须确保在目标空间被填满之前，来源空间中的存活对象全部都完成迁移，这样才能确保一个回收周期的结束。我们约定：在最差情况下，来源空间中需要迁移的存活对象总量（以需要处理的字节数计算）和需要为高优先级线程初始化的内存总量（以满足其在一个回收周期内的内存分配需求）之和为 W_{\max} ；翻转完成之后，至少有 F_{\min} 字节的内存得到释放且可以重新用于分配。也就是说， W_{\max} 表示在一个回收周期内回收器需要处理的最大工作量， F_{\min} 表示回收周期结束后回收器至少要释放的内存量。因此，我们将最小回收率（minimum GC ratio, GCR_{\min} ）定义为：

$$GCR_{\min} = \frac{W_{\max}}{F_{\min}}$$

当前回收率（current GC ratio, GCR）为回收器所执行的回收工作量 W 与目标空间中新分配对象总量 A 的比值，即：

$$GCR = \frac{W}{A}$$

赋值器的分配操作会增加 A 的值，而回收工作则会增加 W 的值。回收器必须完成足够多的回收工作，以确保当前回收率不小于最小回收率，即 $GCR \geq GCR_{\min}$ 。这一条件可以确保即使是在最差情况下，来源空间也可以得到清空（即所有存活对象都得到迁移）。

由于高优先级回收任务的存在，低优先级任务的内存分配速度可以在一定程度上得到控制，从而确保当前回收率 GCR 不会太低（即低于 GCR_{\min} ）。分配过程需要处理的回收工作上限正比于新分配对象的大小。

如果某一高优先级任务在回收器即将进行半区翻转之前得到激活，则目标空间中的剩余内存可能会无法同时容纳高优先级任务所分配的最后一个对象以及最后一个需要从来源空间中迁出的对象。因此，回收器必须确保在 `bottom` 和 `top` 之间有足够大的空间来容纳这些对象，确切地讲，这块空间必须大到足以容纳高优先级任务在当前回收周期结束之前新分配的所有对象。为达到这一目的，应用程序的开发者必须事先评估出最差情况下高优先级任务的内存分配需求、其执行周期以及每个周期的最差执行时间。Henriksson 认为，由于控制系统中的高优先级任务通常快速、短小，且其几乎没有内存分配需求，所以这一评估任务对于开发者来讲难度很低。Henriksson 还提出一种用于确定程序可调度性以及为高优先级任务预留内存总量的分析框架，该框架的输入参数为程序的各种执行参数，例如任务时限、任务周期等。

19.4.2 执行开销

高优先级任务的回收相关开销存在严格上界，即内存分配、指针解引用、指针存储操作所需的指令数量都是有限的。当然，仅仅凭借指令数量来评估执行时间通常并不可靠，因为处理器可能还会受到高速缓存不命中等问题的影响。最差执行时间分析要么必须假定所有高

速缓存都不可用（从而拖慢所有加载操作的执行速度），要么必须从经验出发，确保系统能够在预定负载下满足时限要求。

堆访问操作的额外开销为一条转发指针访问指令，外加关中断操作的开销。最差情况下的指针写操作开销包括对目标对象进行标记，以便其在后续过程中得到迁移，这一过程大约需要 20 条指令。分配过程只需要简单地移动阶跃指针并初始化对象头部（包括设置转发指针以及其他头部信息），其执行开销大约为 10 条指令。

低优先级任务的堆访问操作以及指针存储操作也存在相同的开销。在最差情况下，分配过程需要执行的回收工作量正比于新分配对象大小。分配操作所能遭遇的最差情况取决于对象可能占用的最大空间、堆空间大小、最大存活对象集合、给定回收周期内的回收工作量上限。

高优先级任务的最差延迟取决于回收器完成（或者中止）原子操作所需的时间，这一时间通常较短且存在上界。Henriksson 指出，与完成原子操作的时间相比，执行延迟在更大程度上取决于系统完成上下文切换所需的时间。

390

19.4.3 开发者需要提供的信息

为确保回收工作不会干扰高优先级任务的执行，开发者必须为回收器提供足够多的信息以反映应用程序以及高优先级任务的特征，回收器可以根据这些信息计算出最小回收率，同时在程序执行过程中跟踪当前回收率。每种高优先级任务的执行周期以及最差执行时间是回收器必须掌握的信息，除此之外，还要掌握它们在一个执行周期内的最大内存需求量，回收器据此才能计算出满足高优先级任务分配要求的最小内存量。开发者同时还必须对应用程序的最大存活内存量进行评估。通过上述各项参数，便可进一步针对高优先级实时任务进行程序的最差执行时间分析、可调度性分析。Henriksson[1998] 提供了更多细节。

19.5 基于时间的实时回收：Metronome 回收器

基于间隙的调度策略要求高优先级实时任务之间存在足够长的间隙来执行回收任务，而基于时间的调度（time-based schedule）策略则将最小赋值器使用率作为调度问题的一个输入参数，也就是说，调度器不仅要确保实时任务满足系统的时限要求，也要保证系统的最小赋值器使用率。基于时间的调度策略首先在面向 Java 的 Metronome 实时垃圾回收器中得到应用 [Bakon 等, 2003a]，它是一个增量式的标记-清扫回收器，并会在必要时进行局部整理以避免堆的碎片化^①。该回收器使用删除写屏障来维护弱三色不变式，即回收器会将写操作所覆盖的指针域的目标对象标记为存活。标记过程中新分配的对象标记为黑色。与 Blleloch 和 Cheng[1999] 的副本复制回收器相比，该回收器写操作的标记开销更低（同时也具有更高的可预测性）。

在完成清扫并将垃圾回收之后，Metronome 回收器会视情况决定是否需要进行整理，整理的目的在于确保在下一个回收周期结束之前，堆中存在足够多的连续空闲内存来满足赋值器的分配需求。与 Henriksson [1998] 类似，Metronome 回收器也使用 Brooks 式转发指针，尽管它会给赋值器的每次堆访问操作带来额外的间接负担。

① Metronome 的中文含义为“节拍器”，象征着赋值器和回收器之间的时间分配可以像节拍器一样精确——译者注。

19.5.1 赋值器使用率

Metronome 回收器可以确保赋值器在整个系统的执行时间中至少占有某一预定的比例，而其他执行时间则可以由回收器自由支配，即回收器可以将自己不需要的执行时间让给赋值器。Metronome 回收器能够确保回收停顿时间的均匀性、短暂性，进而能够为赋值器提供比传统回收器更细粒度的使用率保障。Metronome 回收器的默认设置是：在 10ms 内，以 500 μ s 作为最小时间单元，确保最小赋值器使用率达到 70%，也可以对这一目标进行调整，以便进一步满足系统的空间限制。图

19.6 展示了 Metronome 回收器的一个为期 20ms 的回收周期，回收器将系统的执行时间划分为 500 μ s 的时间片，同时确保在任意一个时长为 10ms 的滑动窗口内，赋值器回收率均不低于 70%，即：在任意一个 10ms 的时间窗内，回收器最多占据 6 个时间片，而赋值器则至少占据 14 个时间片。即使调度器连续为回收器分配两个时间片，系统仍能够达到预定的最小赋值器使用率，但为了最大限度地降低停顿时间，调度器在为回收器分配一个时间片之后，紧接着会为赋值器分配至少一个时间片，从而确保回收停顿时间不会超过一个时间片。如果允许最小赋值器使用率小于 50%，则在某一时间窗口内，调度器分配给回收器的时间片就有可能超过赋值器，此时调度器便有可能为回收器分配两个连续的时间片。

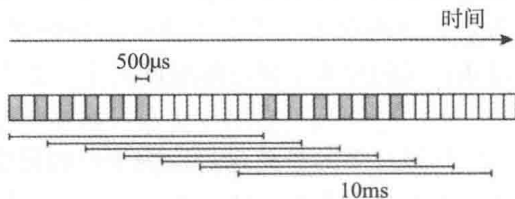


图 19.6 Metronome 回收器中的时间片分配情况。灰色单元为回收器占用的时间片，白色为赋值器占用的时间片

391

当回收器尚未激活时，赋值器便可占据所有的时间片，其使用率将达到 100%。因此，赋值器使用率可能会由于回收器的执行而出现周期性的下降，但无论如何也不会低于预设的最小赋值器使用率。图 19.7 从宏观角度展示了每个回收周期中赋值器使用率的下降。

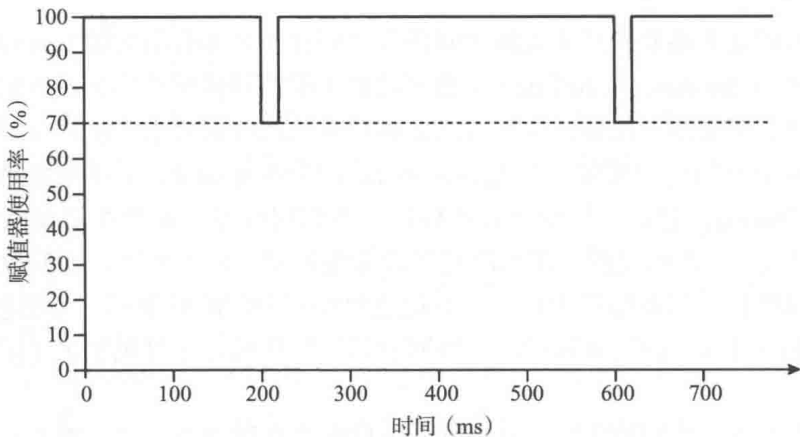


图 19.7 Metronome 回收器的整体赋值器使用率

图 19.8 展示了图 19.6 所示回收周期内任意时刻的赋值器使用率（灰色条带表示回收器所占用的时间片，白色条带为赋值器的）。对于 x 轴上的任意时间点 t ，其所对应的值表示以 t 作为起点的、长度为 10ms 的时间窗内的赋值器使用率。需要注意的是，图 19.6 中的调度器可以保证回收周期内的赋值器使用率精确达到 70%，而真正的调度器却通常无法达到这样

392

的精确。实际应用中的调度器通常会在赋值器使用率接近预定的最小值时进行回调，从而避免赋值器使用率低于这一目标。

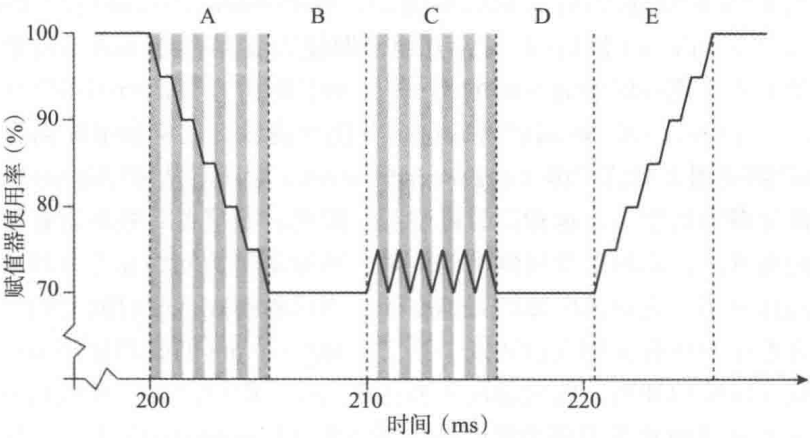


图 19.8 Metronome 回收器在一个回收周期中的赋值器使用率

首次发表于 IBM developerWorks: <http://www.ibm.com/developerworks>

在图 19.8 中，区域 A 中的曲线呈阶梯状，其中的下降部分是由回收器时间片导致的，而平坦部分则归功于赋值器所获取的时间片。阶梯状的下降趋势表明，回收器会与赋值器交替执行，以降低回收停顿时间，同时赋值器使用率也会逐渐趋近预定的最小值。区域 B 中仅有赋值器处于活动状态，其目的是确保所有包含该区域的滑动窗口的赋值器使用率均满足要求。在该区域中我们可以看到，回收器仅会在滑动窗口的起始部分处于活动状态，其原因在于，回收器会在满足停顿时间和赋值器使用率的前提下尽量贪心地占用时间片。这同时也意味着回收器会在滑动窗口的起始部分耗尽其最大可能占用的时间，并将所有其他时间留给赋值器。区域 C 展示了当赋值器使用率接近目标值时回收器的活动状态，曲线的上升部分意味着调度器察觉到赋值器使用率接近预定目标值，并将时间片分配给了赋值器，而曲线的下降部分则意味着调度器将时间片分配给了回收器，从而对赋值器使用率朝着趋近于目标值的方向进行调整。锯齿状的赋值器使用率曲线表明，通过与赋值器交替执行的方式，回收器不仅可以实现短时停顿，还可以满足预定的赋值器使用率要求。区域 D 中的曲线表明，回收周期结束后，赋值器仍需运行一段时间，才能确保赋值器使用率开始反弹。区域 E 中，赋值器使用率从目标值阶梯状地上升到 100%。

19.5.2 对可预测性的支持

在确保回收器安全性的前提下，Metronome 回收器使用多种技术来保证系统停顿时间的可预测性。第一项技术所针对的问题是：当赋值器在碎片化的堆中分配大对象时如何确保分配耗时的可预测性。其他技术则用于确保回收停顿时间的短暂性，从而进一步提高停顿时间的可预测性。

子数组。Metronome 回收器支持子数组，即赋值器可以在多个内存块中分配数组，这一策略能够容忍堆达到一定的碎片化程度，从而减少了对整理操作的依赖（进一步增强了可预测性）。大数组将以单块连续索引对象 (spine object) 的形式分配，真正的数组元素将保存在独立分配的、大小固定的子数组中，而索引对象则用于记录这些子数组的指针。子数组的大小为 2 的整数次幂，因而对数组进行索引操作所需的除法操作可以通过移位操作来实现。借

助于索引对象这一中间层，我们可以简单地计算元素在数组中的位置。Metronome 回收器将子数组的大小设置为 2KB，而索引对象的最大尺寸为 16KB，因而数组最大可以达到 8MB。

读屏障。与 Henriksson[1998] 式回收器类似，Metronome 回收器也使用 Brooks 式读屏障来确保赋值器在访问对象时只存在一元开销，即使回收器已经移动了对象。研究者们曾经认为，以软件方式实现读屏障的开销过大——Zorn[1990] 在其运行时系统中的开销测量结果为 20% 左右——但 Metronome 回收器通过多种优化技术可以将读屏障的平均开销降低到 4%。第一，回收器使用立即读屏障（eager read barrier）对所有从堆中加载的引用进行转发，从而确保赋值器加载到的引用永远指向目标空间，因此，赋值器对栈和寄存器中引用的访问便不存在任何间接开销，而如果使用懒惰读屏障，则赋值器访问栈和寄存器中的引用时就必须付出额外的间接开销。立即读屏障的开销在于，当回收器在回收时间片内移动对象时，其必须对栈和寄存器中的所有引用进行转发。第二，Metronome 回收器使用多种通用的编译器优化技术来降低读屏障的开销，例如通用子表达式消除，除此之外，还包括一些特殊的优化技术，例如将屏障移动到真正需要使用时的屏障下沉（barrier sink）技术、将屏障与空指针检测相结合的技术 [Bacon 等，2003a]。

回收器的调度。Metronome 回收器使用两个不同的线程来控制持续调度以及短时不可中断停顿。闹钟线程（alarm thread）拥有很高的调度优先级（高于所有赋值器线程），并会以 500 μ s 为单位周期性地得到唤醒。该线程以这种“心跳”的方式来判定是否要对回收器进行调度，如果需要，它会初始化赋值器线程的挂起流程（但不执行挂起操作——译者注），并唤醒回收器线程。闹钟线程的活动时间仅足以完成这些任务（通常可以在 10 μ s 内完成），因而它几乎不会对应用程序造成任何影响。

回收线程（collector thread）会在每个回收器时间片中承担真正的回收工作。该线程必须首先完成闹钟线程已经初始化的赋值器线程挂起任务，然后才能利用剩余的时间片来执行回收相关工作。如果回收器发现自己无法在时间片结束之前完成工作，则其可以提前进入睡眠状态。

由于赋值器和回收器以固定的时间交替执行，所以 Metronome 回收器可以获得固定的 CPU 使用率。但是，如果赋值器的分配耗时会发生变化，则基于时间的调度策略很容易受到不同内存分配需求的影响。

赋值器线程的挂起。Metronome 回收器使用一系列较短的增量停顿来完成每个回收周期内的回收任务。但其同时也必须为每个回收器时间片挂起所有赋值器线程，并使用某种握手机制来确保所有赋值器线程都挂起在安全回收点。此时，每个赋值器线程将释放所有持有的运行时元数据、将其当前上下文中的所有引用存储到预定义位置、发出信号以表明其已经到达安全回收点，然后进入睡眠并等待唤醒信号。得到唤醒之后，每个线程会重新为其上下文加载对象指针，重新获取其曾经持有的、必要的运行时元数据，然后再继续执行。由于赋值器线程在挂起/恢复时会进行存储/重新加载对象指针的操作，所以回收器可以在回收时间片内更新其中指向已迁移对象的指针。编译器会在赋值器执行代码中有规律地、间隔性插入安全回收点，从而确保挂起赋值器线程所需的时间存在上界。

上述赋值器线程挂起机制仅针对处于活动状态的赋值器线程，不访问堆的线程、执行非赋值器“本地”代码的线程、已经挂起的赋值器线程（如出于同步目的等待）都无需挂起。在回收器执行的过程中，如果线程开始执行（或者返回到）可以操作堆的赋值器代码（例如从“本地”代码中返回、调用 Java 原生接口的相关操作、访问其他 Java 运行时数据结构等），

则其需要将自身挂起并等待回收器时间片执行完毕。

非协同 (ragged) 根扫描。Metronome 回收器会在一个时间片内完成所有赋值器线程栈的扫描, 其目的在于避免出现指针丢失问题。因此, 开发者必须避免在其实时应用程序中使用过深的函数调用, 从而确保回收器可以在一个时间片内完成线程栈的扫描。尽管每个栈的扫描都必须在一个时间片内原子化地完成, 但 Metronome 回收器依然允许在不同的时间片内扫描不同赋值器线程的栈, 也就是说, 即使回收器正在扫描某个线程的栈, 系统也允许回收器和赋值器线程交替执行。为达到这一目的, Metronome 回收器需要为所有尚未扫描的线程设置一个写屏障, 该屏障可以确保这些线程不会在回收器对其进行扫描之前将某一根对象隐藏到回收波面之后。

394

19.5.3 Metronome 回收器的分析

Metronome 回收器的最大贡献之一在于, 其最早提出了垃圾回收调度问题的形式模型, 以及从赋值器使用率和内存使用率角度出发来设计回收器的方法 [Bacon 等, 2003a]。该模型通过如下几个参数实现参数化: 赋值器瞬时分配率 $A^*(\tau)$ 、赋值器瞬时垃圾生成率 $G^*(\tau)$ 、垃圾回收处理率 P (依照存活数据进行测量)。所有这些参数均按照单位时间内的单位数据量进行定义。此处的时间 τ 是指赋值器时间, 并假定回收器可以运行得无限快 (也可更加实际地假定可用内存足够多, 无需进行垃圾回收)。

通过这些参数, 我们便可简单地将时段 (τ_1, τ_2) 中的内存分配量定义为:

$$\alpha^*(\tau_1, \tau_2) = \int_{\tau_1}^{\tau_2} A^*(\tau) d\tau \quad (19.1)$$

类似地, 该时段内的垃圾生成量可以定义为:

$$\gamma^*(\tau_1, \tau_2) = \int_{\tau_1}^{\tau_2} G^*(\tau) d\tau \quad (19.2)$$

Δt 时段内的最大内存分配量为:

$$\alpha^*(\Delta\tau) = \max_{\tau} \alpha^*(\tau, \tau + \Delta\tau) \quad (19.3)$$

进一步得出最大内存分配率^①为:

$$a^*(\Delta\tau) = \frac{\alpha^*(\Delta\tau)}{\Delta\tau} \quad (19.4)$$

在给定时间 τ , 程序的瞬时内存需求量 (包括垃圾、额外开销、内存碎片) 为:

$$m^*(\tau) = \alpha^*(0, \tau) - \gamma^*(0, \tau) \quad (19.5)$$

程序的真正执行时间当然还要包括回收器的执行时间, 因而我们引入函数 $\Phi: t \rightarrow \tau$ 来表示从真正时间 t 到赋值器执行时间 τ 的映射, 此时必然有 $\tau \leq t$ 。我们将以赋值器时间作为参数的函数记为 f^* , 而将以真正时间作为参数的函数记为 f 。则在 t 时刻, 程序的存活内存总量为:

$$m(t) = m^*(\Phi(t)) \quad (19.6)$$

而程序在整体执行过程中的最大内存需求量为:

$$m = \max_{\tau} m(t) = \max_{\tau} m^*(\tau) \quad (19.7)$$

时间使用率。除了上述各项参数之外, 基于时间的调度策略还包括两个额外的参数: 赋

① 此处需要注意区分 a^* (某一时段内的最大内存分配率) 与 α^* (程序整个生命周期内的最大内存分配量)。

值器执行时间单元 Q_T 以及回收器执行时间单元 C_T ，它们分别表示赋值器和回收器在放弃执行 (yield) 之前可以使用的时间量。据此，我们可以把最小赋值器使用率定义为：

395

$$u_T(\Delta t) = \frac{Q_T \cdot \left[\frac{\Delta t}{Q_T + C_T} \right] + x}{\Delta t} \quad (19.8)$$

其中， $Q_T \cdot \left[\frac{\Delta t}{Q_T + C_T} \right]$ 表示在给定时段内赋值器执行时间单元的总和， x 表示其余的赋值器执行时间单元，其定义是：

$$x = \max \left(0, \Delta t - (Q_T + C_T) \cdot \left[\frac{\Delta t}{Q_T + C_T} \right] - C_T \right) \quad (19.9)$$

随着程序的执行，最小赋值器使用率会逐渐趋近于赋值器占整体执行时间的期望值，即：

$$\lim_{\Delta t \rightarrow \infty} u_T(\Delta t) = \frac{Q_T}{Q_T + C_T} \quad (19.10)$$

我们以一个可以实现精确调度的系统为例，假定回收器执行时间单元 $C_T = 10$ ，即赋值器可能经历的最大停顿时间为 $10 \mu s$ 。图 19.9 展示了在赋值器执行时间单元分别为 $Q_T = 2.5$ 、 $Q_T = 10$ 、 $Q_T = 40$ 时的最小赋值器使用率曲线。我们注意到：当 Δt 足够大时， $u_T(\Delta t)$ 逐渐向 $\frac{Q_T}{Q_T + C_T}$ 收敛；回收器的执行频率越高（即赋值器执行时间单元 Q_T 越短），则曲线的收敛速度越快。除此之外，时间范围越小，则参数 x 对实时系统的影响比重越大。当然，实际应用中的回收器通常只会断续执行，因而 $u_T(\Delta t)$ 只是赋值器使用率的下限。

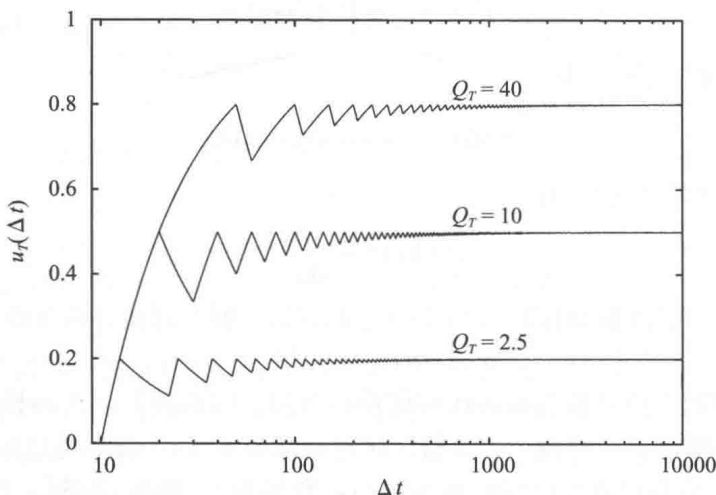


图 19.9 在一个得到精确调度的基于时间的系统中，最小赋值器使用率 $u_T(\Delta t)$ 的变化曲线。 $C_T = 10$ ，

赋值器使用率逐渐向 $\frac{Q_T}{Q_T + C_T}$ 收敛。提升回收器的执行频率（即减少赋值器执行时间单元的时间）可以加快曲线的收敛速度

396

空间使用率。我们曾经提到，空间利用率会根据赋值器分配率的变化而有所不同。假定回收率固定为 P ，则在 t 时刻，回收器将花费 $m(t) / P$ 的时间来处理 $m(t)$ 的存活数据（工作量正比于标记存活对象所需的追踪工作）。回收器每执行一个 C_T 的时间单元，赋值器都会执

行一个 Q_T 的时间单元。因此在时刻 t 执行增量回收所必需的额外空间为：

$$e_T(t) = \alpha^* \left(\phi(t), \phi(t) + \frac{m(t)}{P} \cdot \frac{Q_T}{C_T} \right) \quad (19.11)$$

我们进一步定义所需额外空间的最大值：

$$e_T = \max_t e_T(t) \quad (19.12)$$

Metronome 回收器释放一个垃圾对象可能需要长达三个回收周期：第一个回收周期将判断该对象是否为垃圾；如果其在当前回收周期的快照获取完毕之后立即成为垃圾，则其只能在下一个回收周期中得到回收。而在第二个回收周期中，如果该对象在得到释放之前又需要进行迁移，则其只能在第三个回收周期中得到回收。

因此，系统在时刻 t 所需的空间为（忽略内部碎片）：

$$s_T(t) \leq m(t) + 3e_T \quad (19.13)$$

而整个程序的空间需求量为：

$$s_T \leq m + 3e_T \quad (19.14)$$

这一数值即为系统在最差情况下的空间需求量，此时所有垃圾对象都将被保留到下一个回收周期，而它们在下一个回收周期中又都需要移动。空间需求量的期望值是 $m + e_T$ 。

变更操作。由于写屏障必须记录赋值器所删除和插入的每个引用，所以变更操作也存在额外的空间开销。回收器必须确保写屏障的开销是一个常量。写屏障不仅过滤掉 `null` 引用，而且要对目标对象进行标记以避免重复记录，从而确保回收器的工作量存在上限（最差情况下，堆中所有对象都将被标记为存活）。因此在最差情况下，写屏障日志中的条目数量会与堆中对象的数量相等。针对这一情况，日志条目的分配应当与一般对象的分配有所区分。

敏感性。只有当开发者所提供的参数能够精确反映应用程序与回收器的特征时，Metronome 回收器的运行时行为才能达到预期目标，这些参数包括：应用程序的分配率 $A^*(t)$ 、垃圾生成率 $G^*(t)$ 、回收器处理率 P 、赋值器和回收器各自的执行时间单元 Q_T 和 C_T 。赋值器使用率 u_T 仅取决于 Q_T 和 C_T ，因而其值可以保持稳定（除此之外，仅可能受到操作系统传递时间单元相关信号的波动以及最小时间单元的影响）。

整个程序的空间需求量 s_T 会受到垃圾回收所必需的额外空间 $e_T(t)$ 的影响，而后者又取决于程序的最大内存使用量 m 以及赋值器在一个执行时间单元里的内存分配量。如果开发者低估了总空间需求量 m 或者最大分配率 a^* ，则总内存需求量 s_T 可能会出现不可控制的增长。如果在赋值器某一时刻的分配率过高，则基于时间的回收器很容易出现内存需求量暴增的情况。类似地，开发者也必须对回收器处理率 P 进行较为保守的估计（即小于真正的回收处理率）。

幸运的是，相对于赋值器执行时间单元而言，一个回收周期的持续时间相对较长：

$$\Delta\tau = \frac{m(t)}{P} \cdot \frac{Q_T}{C_T}$$

因此，回收周期内的分配率将接近于平均分配率，因而只要最大内存需求量 m 得到准确评估，系统的空间消耗量将几乎保持不变。

与基于工作的调度策略的比较。我们可以对基于工作的调度策略进行简单的分析，然后再将其与基于时间的调度策略进行比较。但是，赋值器的操作却会影响其能占用的执行时

间, 从而对分析结果造成影响。更加正式地讲, 对于基于时间的调度策略, 从真正时间 t 到赋值器时间 τ 的映射函数 Φ 是线性且固定的, 而对于基于工作的调度策略, 这一映射函数则是变化的、非线性的, 具体取决于应用程序。

在基于工作的调度策略中, 当赋值器达到一定的内存分配量时, 调度器便会唤起回收器并执行一定的回收工作, 这一工作模式可以从回收器的两个输入参数中得到体现: 赋值器工作单元 Q_w 和回收器工作单元 C_w 分别表示调度器允许赋值器 / 回收器在让出 CPU 之前执行多少 (相对) 内存分配 / 回收工作。

对于基于工作的调度策略, 由于其时间映射函数 Φ 是变化的、非线性的, 所以我们无法得出最小赋值器使用率的封闭解 (closed-form solution)。在每个回收增量中, 回收器会以速率 P 处理总量为 C_w 的内存, 因而回收停顿时间固定为 $d = C_w / P$ 。每个赋值器执行单元会分配总量为 Q_w 的内存, 因此第 i 个赋值器执行单元的最小执行时间 $\Delta \tau_i$ 为满足如下等式的解:

$$\alpha^*(\Delta \tau_i) = iQ_w \quad (19.15)$$

增大时间间隔并不会降低赋值器在该时段内的最大内存分配量, 因而 $\alpha^*(\Delta \tau_i)$ 会呈现单调递增的趋势。因此 $\Delta \tau_i > \Delta \tau_{i-1}$, 则等式 (19.15) 可以通过迭代的方式求解。假定 k 为最大的整数, 则有:

$$kd + \Delta \tau_k \leq \Delta t \quad (19.16)$$

因此, 在时段 Δt 内的最小赋值器使用率为:

$$u(\Delta t) = \frac{\Delta}{\Delta t} + \quad (19.17)$$

其中, $\Delta \tau_k$ 为时段 Δt 内 k 个赋值器执行单元的总时长, y 为其余赋值器执行单元, 其定义如下:

$$y = \max(0, \Delta t - \Delta \tau_k - (k+1) \cdot d) \quad (19.18)$$

需要注意的是, 当 $\Delta t < d$ 时, 最小赋值器使用率 $u_w(\Delta t)$ 将低至零。除此之外, 一旦赋值器分配了大小为 nQ_w 的对象, 回收器便必须执行 n 个回收工作单元, 从而产生至少 nd 的回收停顿时间, 在此期间赋值器使用率也将降低至零。这一分析结果表明, 开发者必须避免在基于工作的垃圾回收环境下分配过大的对象, 同时必须确保分配操作分布均匀, 只有这样才能确保应用程序满足实时要求。

最小赋值器使用率取决于赋值器的分配率 $\alpha^*(\Delta t)$ (其中, $\Delta t \leq \Delta t$) 以及回收器处理率 P 。假定需要满足实时性能要求的时段 Δt 较小 (例如 20ms), 则该时段内的峰值分配率可能会很高。因此, 从实时尺度来看, 基于工作的调度策略的最小赋值器使用率 $u_w(\Delta t)$ 将会与赋值器分配率存在很大差异。而对于基于时间的调度策略, 回收器受分配率影响的时间 $\Delta \tau$ 则是一个更大的范围, 即回收器完成一个垃圾回收周期所需的时间。

在空间方面, 回收器在时刻 t 所需的额外空间量为:

$$e_w(t) = m(t) \cdot \frac{Q_w}{C_w} \quad (19.19)$$

进一步得出一个完整的回收周期所需的额外空间总量为:

$$e_w = m \cdot \frac{Q_w}{C_w} \quad (19.20)$$

只要开发者对总存活内存量 m 预估准确, 则该值也必然是准确的。另外需要注意的是,

如果 $Q_W < C_W$, 则一个完整回收周期所需的额外空间总量 e_W 将超过赋值器所需的空间总量 m 。因此在时刻 t , 应用程序的总内存需求量为:

$$s_W(t) \leq m(t) + 3e_W \quad (19.21)$$

则总体空间需求量为:

$$s_W = m + 3e_W \quad (19.22)$$

综上所述, 对于基于工作的回收调度策略, 只要存活内存总量 m 预估准确, 则应用程序必然可以满足空间界限要求, 但其最小赋值器使用率则严重依赖于赋值器执行实时任务时的分配率。与此相比, 尽管基于时间的回收器很容易满足最小赋值器使用率的要求, 但其空间需求总量却可能产生波动。

19.5.4 鲁棒性

基于时间的回收调度策略可以满足实时垃圾回收的鲁棒性 (robustness) 要求, 但是, 一旦回收器的输入参数不够精确, 则其很可能无法回收到足够的内存, 此时唯一优雅的降级策略是降低赋值器的分配率。

一种降低整体分配率的策略是使用分代策略, 即把年轻代作为降低主体堆内存分配率的一个过滤器。将垃圾回收的主要精力集中在堆中最容易生成空闲内存的部分, 不仅能够提升赋值器使用率, 而且有助于减少浮动垃圾。但是, 传统的新生代回收在两个方面存在不确定性, 一是回收所需的时间, 二是需要提升的数据量。Syncopation 回收器是一种可以将新生代回收与成熟空间回收同步进行的策略, 其中新生代存活对象将在成熟空间回收周期的开始阶段以及清扫过程启动时得到提升, 且这一过程均位于成熟空间回收周期之外 [Bacon 等, 2005]。该策略需要对分代回收过程进行分析, 将其新生代存活率作为回收器的输入参数, 并合理设置新生代的大小, 以确保其中所有存活对象可以在一个实时窗口内完成提升。对于任意给定的应用程序, 这一分析方法可以确定分代回收策略是否适用。在程序执行过程中, 赋值器分配率的短时剧烈波动可能导致回收器无法在满足实时要求的前提下完成新生代的提升, 而 Syncopation 回收器的解决方案则是将由此引发的回收工作推迟。Frampton 等 [2007] 采用了另一种策略, 他们允许回收器增量式地执行新生代回收, 从而避免提升年轻代存活对象的耗时过长。

减缓赋值器分配率的另一种策略是简单地引入基于工作调度策略的相关方法, 但这却可能导致赋值器无法满足时限要求。而基于间隙的调度策略则可以通过抢占低优先级赋值器线程的方式来赶上赋值器的分配率, 因此只要高优先级实时任务之间的间隙足够多, 则赋值器依然能够满足实时要求。基于这一观察结果, 研究者们进一步提出了将基于间隙的调度策略与基于时间的调度策略相结合的一整套方法论, 即税收与开支 (tax-and-spend)。

19.6 多种调度策略的结合: “税收与开支”

Metronome 回收器需要将赋值器挂起一小段时间, 并在这段时间内执行一个回收增量, 其能够在专用单处理器或者核数较少的多处理器系统上表现出最佳性能。相比之下, 基于工作的回收器所产生的停顿时间则可能会比基于时间的调度策略多出几个数量级。Henriksson 的基于间隙的调度策略最适用于周期性执行的应用程序, 而一旦高优先级任务的执行间隙不足, 则系统很容易出现过载。为解决不同调度策略各自的局限性, Auerbach 等 [2008] 综合了基于工作的调度策略、基于间隙的调度策略以及基于时间的调度策略, 并提出了一种通用

的垃圾回收调度方法论：“税收与开支”。将该技术应用到 Metronome 回收器之后，其平均吞吐量提升了 10% ~ 20%（时间窗缩小到原有时间窗的 $\frac{1}{4}$ ，响应延迟降低了 3 倍）。

“税收与开支”回收器的基本理念是：每个赋值器线程都应当参与一定的回收工作（即“纳税”），同时也应当与回收器交替执行，以确保满足最小赋值器使用率的要求。回收器也可在赋值器的执行间隙尽量多地执行回收工作，即尽量多地积攒“存款”（credit）以供赋值器“支出”，从而达到减少赋值器的回收工作量、保持或者提升赋值器使用率的目的。

当线程执行到安全回收点时，某些全局要素将决定其是否需要“纳税”，与此同时，赋值器线程在内存分配慢速路径（即当线程本地分配缓冲区耗尽时）中也会判断是否需要将回收工作挂起，判定结果同样取决于赋值器的工作情况（通过如下参数进行分析：内存分配的单元、线程执行时间、已经执行过的安全回收点、物理时间或者某种虚拟时间）。

19.6.1 “税收与开支”调度策略

我们知道，最小赋值器使用率的推导对于开发者而言相对简单，他们只需要认为系统的执行速度要比原生处理器慢，且在最差情况下的响应能力最多只会逼近垃圾回收器的量化限制。最大停顿时间也是衡量回收器对赋值器扰动的指标，但最小赋值器使用率显然要优于前者，因为它会考虑到由于单个短时停顿的聚集而导致的响应超限以及由此导致的执行速度减慢。“税收与开支”调度策略允许不同线程拥有不同的赋值器使用率，从而可以在线程分配率差异较大时保证一定的弹性，也可确保具有严格时限要求的线程尽量少地被中断。另外，位于空闲处理器上的后台线程可以为赋值器线程减少一定的回收工作，从而确保较高的赋值器使用率。回收器可以根据应用程序的特征来选择最佳时间测量方式，既可以是物理时间，也可以基于虚拟时间。任何针对应用程序的分析都必须综合单个线程的实时要求来获取程序的整体行为状态。

每个线程的调度。为控制每个线程的赋值器使用率，“税收与开支”调度策略必须依照每个线程的服务指标来进行回收工作的测量以及调度，同时能够把每个回收增量指派给特定的赋值器线程。回收器可以记录哪些线程完成了哪些工作（包括扩展赋值器操作日志、初始化已分配页，以及其他簿记行为的开销），从而避免为某一线程调度过多的回收工作。

另外，在赋值器线程主动陷入操作系统之前（例如执行分配慢速路径、执行 IO 调用，或者执行不会访问堆的本地代码），为避免操作系统调度器感知到线程用完其操作系统时间片并用其他一些无关线程将其替换，“税收与开支”调度策略会令该线程执行一定的回收增量。这一策略在负载较重的系统中尤为重要。如果令赋值器操作与回收器操作在同一个操作系统线程上交替执行，则操作系统调度器便几乎不会干扰到垃圾回收工作的调度。

当系统中各线程的内存分配率存在显著差别，或者希望高优先级线程（例如事件处理线程）尽量少被打扰时，允许不同线程拥有不同的赋值器使用率便显得十分重要。这同时也能够减少时限要求不高的线程所能得到的时间单元，同时承担更多的回收工作，从而达到系统吞吐量的上升。

“税收与开支”调度策略与基于间隙的调度策略的对比。基于间隙的调度策略能够在经典的周期性实时系统中表现良好，但是，一旦系统出现过载且高优先级任务之间没有足够的间隙，则系统的性能会出现急剧下降，这一原因导致其几乎无法应用于队列式、自适应式（即系统尽量精确地计算出结果，但也容许通过降低精度来避免过载）或者交互式实时系统。基于工作的调度策略依照赋值器的分配工作量来对赋值器“课税”，即其所需执行的回收相

关工作正比于其内存分配量，从而确保回收器能够在内存耗尽之前结束当前的回收周期。该策略的问题在于其最小赋值器使用率过低，且回收停顿时间变化很大。基于时间的调度策略依照赋值器使用率来对赋值器“课税”，因此回收器能够利用给定的处理器时间来处理回收相关工作。由于回收器对赋值器的“课税”是持续性的，因而其在过载情况下依然具有较高的鲁棒性，但是，一旦高优先级任务的间隙不足，则赋值器的执行很容易出现不必要的波动，因为只要赋值器使用率满足要求，回收器便可在任意时刻执行回收工作。

“税收与开支”调度策略与基于间隙调度策略的结合。“税收与开支”调度策略采用了一种经济学模型来将不同的调度策略相结合。每个赋值器线程都承担一定的“负税率”，该值将决定其在一段时间内必须承担的回收工作量，进而决定了每个线程各自的最小赋值器使用率。专职垃圾回收线程将以低优先级或者空闲优先级在高优先级任务的间隙执行，并为赋值器积累“存款”。回收器通常会将“存款”存入一个全局“账户”中，当然也可以视情况将其存入多个“账户”中。

所有线程的总体“负税”情况（包括赋值器线程的“负税”以及回收器线程所贡献的“存款”）必须能够确保回收器能够在内存耗尽之前完成回收周期。后台回收线程的数量通常会与处理器的数量相同，这一配置方式可以确保回收线程能够在整个系统的执行平缓期天然得到执行。这些线程执行一系列回收工作单元，每个工作单元都会增加相应“存款”。在实时操作系统中，后台回收线程以低优先级方式运行要比以标准空闲优先级运行要好，如此一来这些线程便可以像其他一些真正执行工作的线程一样得到调度，而不是只能在空闲时段得到调度。这些低优先级实时线程也应当存在一定的休眠时间，从而保证即使是回收工作占据整个处理器，非实时线程也能得到调度。这同时也赋予了系统管理员在必要时登录并杀死失控实时进程的能力。

调度器将根据每个线程所期望的最小赋值器使用率进行调度，其不仅要满足赋值器线程的实时要求，而且还要确保回收器能够正常向前执行。如果某一赋值器线程在执行过程中出现“欠税”情况，则其首先尝试从全局“账户”中扣除等额的“存款”来补偿“欠税”，如若成功，则该线程便可免于“纳税”。只有当后台回收线程无法产生足够的“存款”时，赋值器才需要“纳税”，即使“存款”额度不足赋值器线程的一个“纳税”单元，其所需执行的回收工作量也会比正常情况下少。因此，只要高优先级实时线程的执行间隙足够多，则不仅赋值器可以达到高吞吐量、低延迟的标准，而且回收工作也不会出现延误。“税收与开支”调度策略能够以相同的方式来对待单处理器的执行间隙以及多处理器的富余执行能力。

19.6.2 “税收与开支”调度策略的实现基础

“税收与开支”调度策略所依赖的回收器不仅必须是增量式的（只有这样，基于工作的回收器才能以“课税”的方式工作在赋值器线程上），而且必须是并发式的（只有这样，基于间隙的回收器才能与赋值器并发工作在空闲的处理器上）。为了有效利用多处理器资源，其同时也应当是并行式的（只有这样，基于间隙的回收器才能与基于工作的回收器并发执行）。尽管 Metronome 回收器是增量式的，但是其并非针对并发环境而设计的，原因在于基于时间的调度策略要求赋值器和回收器以精确的时间间隔交替执行，且当回收器执行时赋值器应当处于挂起状态。针对这一情况，“税收与开支”调度策略做出了两项重要改进：第一，回收器线程可以与赋值器线程并发执行，因此便可以简单地利用某些处理器的执行间隙来处理回收工作，与此同时，其他处理器依然可以运行赋值器线程；第二，当系统的负载情况决

定回收器有必要从赋值器“窃取”一部分执行时间时，其可以将回收增量以“课税”的形式施加给赋值器。

如果只依赖并发回收，相当于是将垃圾回收线程的调度任务全权交由操作系统调度器负责，这显然是远远不够的，因为其既不能确保应用程序满足实时系统的时限要求，也不能避免堆耗尽。即使是对于实时操作系统，其依然无法利用应用程序的内存分配模式以及空间需求量信息来决定调度策略。

接下来，我们将介绍“税收与开支”调度策略是如何将 Metronome 回收器扩展成一个即时、并发、并行、增量回收器的。其扩展方式与其他即时并行 / 并发回收器类似，但是为了表述的完整性，我们在此重新进行介绍。

基于非协同时段 (ragged epoch) 的全局握手。所有赋值器线程需要针对当前回收器的工作状态产生一致的认知，为达到这一目的，“税收与开支”调度策略并不需要将所有赋值器线程挂起，而是采用非协同时段协议（所谓非协同时段，是指系统允许各线程处于不同的时段，而不是要求先到达某一时段的线程等待其他线程——译者注），该协议存在多种应用场景。例如，在回收的某些阶段，所有赋值器都必须安装特定的写屏障。除此之外，回收周期的结束也要求所有赋值器线程都清空其本地存储缓冲区。对于赋值器线程安装写屏障、进行结束检测这两种场景，非协同时段协议可以确保所有线程都已经进入新的状态。

非协同时段机制使用一个全局共享时段计数器，每个线程均可以通过原子化地增加该计数器的方式发起一个新的时段。除此之外，每个线程也拥有一个本地时段计数器，线程更新本地计数器的方式是将共享时段计数器的值复制到本地时段计数器中，且该操作只能在安全回收点执行。如此一来，每个线程本地时段计数器的值便不可能大于共享时段计数器。任意线程都可以检查所有线程的本地时段计数器值，并找出其中的最小时段值，该时段即为整个系统的已确认时段 (confirmed epoch)。如果某一线程更新了全局时段计数器，那么只有当已确认时段达到或者超过其所设置的值时，该线程才能确定所有线程都已经注意到这一变化。如果硬件的顺序性保障较弱，则线程在更新本地时段计数器之前必须使用内存屏障。对于陷入 I/O 等待或者执行本地代码的线程，其在返回托管环境时必须首先更新本地时段计数器，然后才能继续执行其他时段敏感操作。如此一来，系统便可简单地将这些线程看作是已经处于当前时段，从而无需等待其返回托管环境。

基于“最后离开者” (last one out) 协议的阶段协商机制。在 Metronome 回收器中，各线程很容易就回收器处于哪一阶段（例如标记、清扫、终结等）达成一致，其原因在于该回收器的回收相关工作是由专门的线程来负责的，只要这些线程的共享回收时间片依然足够，它们便可阻塞式地翻转到下一个状态。但是，如果并发执行的赋值器线程也需要承担一部分回收工作，则每个赋值器线程在进行“纳税”时便可能处在不同的执行阶段，此时的回收阶段检测机制就必须是非阻塞式的，否则需要“纳税”的赋值器线程便有可能无法满足实时响应要求。由于非协同时段握手机制无法将需要“纳税”的赋值器线程与其他赋值器线程相区分，所以其在这一场景下并不高效。与此相比，“最后离开者”协议将阶段指示器与工作者计数器保存在单个共享且可以原子化更新的位置，从而可以高效地实现回收阶段的变更。

每个开始“纳税”的线程都会原子化地增加“纳税者”计数器的值，但其并不会更新阶段指示器。如果某一赋值器线程在完成“纳税”之后发现该阶段还存在剩余工作，其将原子化地减少工作者计数器的值，同时依然不会更新阶段指示器。一旦某个线程发现该阶段（貌

似)没有更多工作要做,且其自身是当前唯一的工作者(即工作者计数器的值为1),则将认为当前阶段可能已经结束,此时,该线程将通过原子操作同时完成回收阶段的变更以及工作者计数器的减少,从而实现回收状态的变迁。

该协议能够正常工作的前提是,每个工作者线程在完成“纳税”之后必须将剩余的未完成工作归还给全局工作队列。垃圾回收工作无论如何最终都会被全部处理,因此必然会存在一个线程成为最后完成工作的线程,该线程将把整个应用程序带入下一个回收阶段。

不幸的是,该策略并不能简单用于 Metronome 回收器标记阶段的结束检测,因为该回收器的删除屏障会将被覆盖的指针添加到每个线程本地更新日志中,而标记阶段结束的必要条件之一便是所有线程的更新日志都为空(注意是所有线程而不只是执行回收工作的线程)。因此,“税收与开支”回收器需要额外引入一个最终标记阶段(final mark phase),该阶段将只允许一个线程来处理回收工作,该线程将使用非协同时段握手机制来判定所有线程的更新日志是否都已经为空。如果判定结果为假,该线程将发出尚未完成的通知,并将回收器的状态切换回并行标记阶段。不论如何,最终标记阶段的结束条件最终都会满足,该阶段的唯一回收线程必然可以将回收器带入下一个阶段。

每个线程回调(per-thread callback)。回收周期内的大多数回收阶段都只需要部分线程参与回收工作即可确保回收器的正常执行,但在其他回收阶段中,回收器却必须要求每个赋值器线程都完成一些工作(或者为每个赋值器线程完成一些工作)。例如,在回收的第一阶段便必须扫描每个赋值器的栈,再如,其他阶段要求赋值器线程刷新其本地缓冲区并将其交由回收器。为满足这一需求,回收器需要在某些回收阶段使用回调协议来替代“最后离开者”协议。

在回调阶段,某个回收器主线程会周期性地检查所有赋值器线程,并判定它们是否执行了回收器指定的工作。如果某一赋值器线程尚未开始执行,回收器会要求其在下一个安全回收点执行某一回调函数来完成某项工作(例如线程栈扫描、缓存刷新等)。而对于陷入 I/O 或者执行本地代码的线程,回收器会阻止其返回托管环境并为其完成相应工作。因此,回调协议的最大时间延迟便是赋值器执行指定操作所需的时间。

优先级提升。毋庸置疑,实时垃圾回收器在堆空间耗尽之前完成回收是确保应用程序正常执行的必要条件。但是,如果高优先级线程长期占据处理器而导致某些低优先级线程无法做出响应,则上述三种协议(非协同时段、最后离开者、回调)都无法正常工作。此时的解决方案是临时性地提升低优先级线程的优先级,直到回收器收到其响应为止。

19.7 内存碎片控制

实时垃圾回收器必须控制时间和空间的消耗上界。不幸的是,随着时间的推移,内存碎片会逐渐侵蚀回收器的空间上界。对内存碎片情况的分析不可避免地需要依赖应用程序的自身行为特征,例如指针密度、平均对象大小、对象大小的局部性等。因此,实时回收器必须能够通过某种方式来管理和限制堆的碎片率。一种显而易见的策略是进行整理,而另一种策略是使用离散分配(以子对象或者子数组的形式进行分配),该方案可以杜绝外部碎片,代价是引入额外(但有限)的内部碎片,同时赋值器的读写操作在访问子对象/子数组时也会存在一定的额外开销。本节我们将分别对这两种策略进行介绍。

实时回收器进行并发整理的挑战在于,如何在确保赋值器访问操作始终满足严格时限要求的前提下实现对象的并发迁移。第 17 章中所介绍的副本复制回收器以及 Blelloch 和

Cheng[1999] 的最初设计明确允许并发复制,但它们均需要为每个对象维护两个副本。在缺乏严格一致性保障的现代多处理器环境下,保持多个副本之间的一致性通常需要使用某种形式的锁,特别是对于 `volatile` 域。除此之外,副本复制回收器需要通过一个同步的终结阶段来确保所有赋值器线程的根都完成转发。单个对象级别的锁通常不会带来较大问题,而对于基于页保护机制的 Compressor 和 Pauseless 回收器而言,其同步级别则是页级别的,赋值器不仅需要付出页保护陷阱的开销,而且回收器的工作模型是基于工作的,除此之外,在阶段切换完成之后,赋值器还会遭遇陷阱风暴问题,因此其最小赋值器使用率通常很低。

如果回收算法不能满足无锁要求,则意味着赋值器的正常执行无法得到保障,更不用说满足系统的时限要求了。接下来,我们将介绍几种并发整理算法,它们均可以保证赋值器的访问操作能够达到无等待或者无锁级别。

19.7.1 Metronome 回收器中的增量整理

Metronome 回收器最初是作为主体非复制式 (mostly non-copy) 回收器来设计的,因为其设计者认为堆中极少出现外部碎片。该回收器使用子对象/子数组来将大对象/大数组拆分为多个内存块,而这些内存块也是应用程序从操作系统中进行分配的最大连续单元。这一策略在很大程度上降低了以减少内存碎片为目的对象复制操作的必要性。Bacon 等 [2003b] 提出了一种分析框架,该框架可以确定每次回收过程需要对多少个页进行碎片整理才能确保赋值器永远不会因为内存分配而陷入等待。由于 Metronome 回收器是增量式的,所以其可以在所有赋值器线程都被挂起时执行碎片整理,而当赋值器线程恢复执行时,它们可以借助于 Brooks 式间接屏障来执行必要的转发工作。与此同时,赋值器永远不会感知到对象复制的中间过程,其唯一的开销便是一层额外的间接屏障,而该屏障在时间上得开销是有界的。“税收与开支”调度策略将 Metronome 回收器扩展为并发回收器,但其并不会执行任何形式的整理操作。

Bacon 等 [2003b] 的分析框架会依照分区适应分配器的空间大小分级 (size class) 来尽量均匀地划分碎片整理工作(由碎片整理目标所决定)。每个空间大小分级将关联一个页链表而非对象链表。该算法对一个空间大小分级进行碎片整理时遵从以下步骤:

- 1) 依照页中存活对象的数量进行排序,从最密集到最稀疏。
- 2) 将首个未填满页(即存活对象密度最高的页)作为分配页(allocation page)。
- 3) 将最后一页(存活对象密度最低的页)作为待整理页。

4) 如果待整理页中的存活对象数量小于某一阈值,且待整理页并非分配页,则回收器将待整理页中的所有存活对象迁移到分配页的空闲内存单元中(如果当前分配页被填满,则填充到下一页中)。

该算法在本质上是将最稀疏页中的对象迁移到密度最高的页,且能够以最小的对象移动代价来将最多的页填满。算法第 2) 步中所选择的分配页是存活对象密度最高的未填满页,这一策略可能会降低应用程序的局部性,因为其可能导致新分配的相关对象散布在多个页中。针对这一问题,我们可以为分配页的填充密度设置一个上限值,从而确保分配页中存在足够的空间,以保持整个程序的局部性。

指向已迁移对象的引用将在后续的追踪阶段得到扫描以及重定向,因此当下一个标记阶段结束时,上一个回收周期的已迁移对象便可得到释放。与此同时,Brooks 式转发屏障可以确保赋值器能够访问到已迁移对象的正确副本。将更新已迁移对象来源引用的操作推迟到

下一个标记阶段存在三方面优势：首先，无需引入额外的“修正”阶段；其次，需要修正的引用数量会更少（已经死亡的对象便无需进行扫描）；最后，将修正操作与追踪操作相结合有助于提升回收器的局部性。

19.7.2 单处理器上的增量副本复制

在进一步介绍更加复杂的并发整理策略之前，我们有必要强调的是，许多实时应用程序都运行在嵌入式系统中，而单处理器又在嵌入式系统中占据支配地位。在单处理器上保持赋值器操作的原子性十分简单（在回收器或者其他赋值器线程看来），只需要简单地禁止调度器中断或者只允许线程在安全回收点进行切换（需要确保所有赋值器屏障的内部代码都不包含安全回收点）。如果在这一环境下使用复制式回收器，只要其可以保证赋值器能够访问已复制对象的唯一副本（使用 Brooks 式间接屏障来强制赋值器满足目标空间不变式），或者确保赋值器能够同时对新老副本进行更新（即副本复制回收，此时赋值器所读取到得仍然是对象的旧有副本），回收器便可自由地进行对象复制。

Kalibera[2009] 对副本复制回收策略与基于 Brooks 式写屏障的复制策略进行了比较，其运行环境为基于单处理器的 Java 实时系统。其副本复制策略依然会以常规的方式为所有对象维护转发指针，唯一的不同之处在于，目标空间新副本的转发指针将指回其来源空间的对象主体（与 Brooks[1984] 不同）。这一策略可以简化赋值器屏障的设计并提升其可预测性：赋值器的 Read 操作无需关心所访问的对象究竟位于来源空间还是目标空间，其可以简单地从任意一个副本中读取数据，而 Write 操作则需要同时更新两个副本以确保它们之间的一致性。算法 19.7 展示了该屏障的伪代码（忽略了对并发扫描的必要支持）。毋庸置疑，免除了每个读操作的转发开销将大幅提升程序的性能，而双写操作的开销在大多数情况下都是微不足道的，因为转发指针通常指向其自身，所以两次写操作的目标地址往往相等。

405

对于多处理器情况下的并发整理，我们很难想到一种直接的方法来实现原子化的 Read 与 Write 操作，因此，接下来我们便必须在考虑赋值器之间、赋值器与回收器之间引入更细粒度的同步。

算法 19.7 单处理器环境下的副本复制算法

```
1  atomic Read(p, i):
2      return p[i]
3
4  atomic Write(p, i, value):          /* p 既可能是对象主体，也可能是其副本 */
5      /* 此处也需要删除屏障相关代码来支持快照回收 */
6      p[i] ← value
7      r ← forwardingAddress(p)
8      r[i] ← value
9      /* 此处也需要插入屏障来支持增量更新回收 */
```

19.7.3 Stopless 回收器：无锁垃圾回收

Pizlo 等 [2007] 在其 Stopless 回收器中提出了一种并发整理算法，即使回收器正在进行并发整理，该算法仍可以确保赋值器操作（包括内存分配以及堆访问操作）达到无锁级别的前进保障。与 Blelloch 和 Cheng[1999] 不同，Stopless 回收器并不要求赋值器同时更新对象的新旧副本以保持其一致性，而是通过某种协议来确保赋值器只需更新一个确定的副本。Stopless 回收器的创新之处在于，其会为每个正在复制的对象创建一个“宽”版本中间

对象，并为该中间对象的每个域都关联一个状态字，而程序则可以使用 `CompareAndSwapWide` 原子操作来同步地复制对象的域。每个域的状态字将与该域的值一起原子化地更新，状态字的值将反映其对应域最新数据所处的位置（位于以下三个位置之一：来源空间的原始副本、宽副本、目标空间中的最终副本）。与 Blelloch 和 Cheng[1999] 类似，该算法也需要在每个对象的头部维护一个 Brooks 式转发指针，该指针将指向宽副本或者目标空间副本。在整理阶段，赋值器线程和回收器线程将通过竞争的方式创建宽副本，具体的竞争方式是使用 `CompareAndSwap` 操作来安装转发指针。

当对象的宽副本创建完成且其转发指针已经指向其宽副本之后，赋值器将只对宽副本进行更新。宽副本中每个域所对应的状态字将（通过读/写屏障）指引赋值器究竟应当读/写哪个副本中的域，状态字的值可能为如下三种：`inOriginal`、`inWide`、`inCopy`。每个状态字的初值均为 `inOriginal`，该值意味着赋值器应当从来源空间的原始副本中读取对应域的值。所有的更新操作都将基于宽副本执行：对于回收器而言，其需要将原始副本中的每个域复制到宽副本中，而对于赋值器而言，其写操作将直接更新宽副本。写操作需要使用 `CompareAndSwapWide` 来完成，该操作不仅会设置宽副本中域的值，还会同时将域的状态字更新为 `inWide`。回收器在更新宽副本中的某个域之前必须确保其所对应的状态字为 `inOriginal`，如果失败，则意味着赋值器已经更新了该域，回收器可以直接跳过对该域的更新。

当某一对象宽副本中所有域的状态都已成为 `inWide` 之后（不论是由回收器完成，还是由赋值器完成），回收器便可在目标空间中为其分配最终的“窄”副本，并将宽副本的转发指针设置为窄副本的引用。此时同一对象将在堆中存在三个版本：来源空间中的旧有对象（其转发指针指向宽副本）、包含最新数据的宽副本（其转发指针指向目标空间副本）、目标空间中尚未初始化的窄副本。回收器会将宽副本中的每个域并发复制到目标空间的窄副本中，对于宽副本每一个刚刚完成复制的域，回收器需要使用 `CompareAndSwapWide` 操作来确保该域并未发生变化，同时将该域所对应的状态字设置为 `inCopy`，如果失败，表示在回收器执行的复制过程中赋值器更新了该域，此时回收器便需进行重试。如果赋值器遇到宽副本中状态字为 `inCopy` 的域，其需要通过转发指针来访问目标空间中的窄副本。

由于 Stopless 回收器能够确保得到更新的域永远是最新的域，所以其能够在不使用锁的前提下支持 Java 的 `volatile` 域。该回收器还能模拟应用程序级别的原子操作，例如赋值器针对特定域的比较并交换操作，其具体细节可以参见 Pizlo 等 [2007]。该策略所遇到的唯一问题是，其无法实现双字域（例如 Java 中的 `long`）及其状态字的原子操作，因为 `CompareAndSwapWide` 操作无法覆盖到双字及其相邻的状态字。针对这一问题，Stopless 的作者提出了一种基于标准 `CompareAndSwapWide` 来模拟 n 路比较并交换操作的技术 [Harris 等，2002]。

读者可能会对 Stopless 回收器的空间开销提出挑战（同一对象存在三个副本，且宽副本占用的空间还要翻倍），但 Pizlo^①指出，只要待整理的稀疏页中最多只填充到三分之一的程度，我们便可利用其中死亡对象的空间来创建临时性的宽副本。由于待整理页原本已经碎片化，所以其中可能没有足够的连续空闲内存来容纳所有对象的宽副本。但是，如果分配器使用分区适应分配策略，则待整理页中的空闲内存将大小一致，此时回收器便有可能利用页中

① 与 Filip Pizlo 的私人沟通。

的连续碎片来分配宽副本，从而确保宽副本中的每个数据域与其状态字相邻。在 Stopless 回收器中，宽副本所占据的空间将会保留到下一轮回收的标记阶段结束之后，此时回收器才能确保所有的指针都已转发到目标空间。

19.7.4 Staccato 回收器：在赋值器无等待前进保障条件下的尽力整理

Metronome 回收器需要在赋值器线程被挂起的回收器时间片中进行整理，而 Staccato[McCloskey 等，2008] 则可以在不使用锁的前提下实现并发整理，且在一般情况下也无需使用诸如 CompareAndSwap 的原子操作，即使是对于内存顺序性保障较弱的多处理器平台也不例外。该算法避免陷入原子操作风暴的策略是仅移动少量对象（仅在回收存活对象较为稀疏的页时才需要）以及随机选择待整理页。

Staccato 回收器继承了 Metronome 回收器的 Brooks 式间接屏障，其同样也在每个对象的头部放置一个转发指针。该回收器同样依赖基于非协同握手的同步机制：赋值器需要以固定的间隔（例如在每个安全回收点）来获知全局状态的变更（对于诸如 PowerPC 等顺序性保障较弱的处理器，赋值器需要先执行内存屏障）。回收器会在转发指针中保留一位以表示其是否正处于复制过程中（Java 对象通常依照字来对齐，因而回收器可以复用指针的最低位）。这一 COPYING 位以及转发指针可以通过 CompareAndSwap 或者 CompareAndSet 原子化地进行修改。当需要移动某一对象时，回收器需要执行如下操作：

- 1) 使用 CompareAndSwap/CompareAndSet 原子化地设置 COPYING 位。由于赋值器在访问转发指针时并不会使用原子操作，所以其可能需要一定时间才能感知到这一变更。
- 2) 等待所有赋值器线程都完成非协同握手，目的是确保所有赋值器都可以感知到 COPYING 位的变化。
- 3) 执行读内存屏障来确保回收器可以感知到赋值器在完成非协同握手之前的更新操作（只有对于顺序性保障较弱的处理器才需如此）。
- 4) 分配副本并将原始对象中的数据复制到其中。
- 5) 执行写内存屏障来确保刚刚写入的数据全局可见（只有对于顺序性保障较弱的处理器才需如此）。
- 6) 发起非协同握手并等待所有赋值器线程都完成握手，赋值器将在应答过程中执行读内存屏障，目的是确保其能够感知到已经写入副本的数据。
- 7) 使用 CompareAndSwap/CompareAndSet 将原始对象的转发指针设置为新副本的地址，同时清空 COPYING 位。该操作相当于是尝试将对象的移动“提交”，如若失败，则意味着赋值器已经在某一时刻修改了对象，此时移动将中止。

回收器通常会尝试移动一批对象，因此基于非协同握手的同步操作的开销将会得到分摊，如算法 19.8 中的 copyObjects 方法所示。该方法的输入参数为待移动对象集合 candidates，而其返回结果则为移动失败的对象集合 aborted。

407

算法 19.8 Staccato 回收器中的复制操作以及（复制过程中的）赋值器屏障

```

1 copyObjects(candidates):
2   for each p in candidates
3                                     /* 设置 COPYING 位 */
4       CompareAndSet(&forwardingAddress(p), p, p | COPYING)
5   waitRaggedSynch(readFence) /* 确保所有赋值器线程均能感知到 COPYING 位的变化 */
6   readFence()                /* 确保回收器能够感知到赋值器在 CAS 操作之前的变更 */

```

```

7   for each p in candidates
8       r ← allocate(length(p))                /* 分配副本 */
9       move(p, r)                             /* 复制数据 */
10      forwardingAddress(r)                    /* 副本的转发指针将指向其自身 */
11      add(replicas, r)                        /* 记录副本 */
12  writeFence()                               /* 刷新写操作以确保赋值器感知到回收器修改的数据 */
13  waitForRaggedSynch(readFence)              /* 确保赋值器能够感知到副本 */
14  for each (p in candidates, r in replicas)
15                                          /* 尝试“提交”副本 */
16      if not CompareAndSet(&forwardingAddress(p), p | COPYING, r)
17          /* 提交失败, 进行处理 */
18          free(r)                             /* 将放弃提交的副本释放 */
19          add(aborted, p)                     /* 记录复制失败的对象 */
20  return aborted
21
22 Access(p):
23     r ← forwardingAddress(p)                /* 加载转发指针 */
24     if r & COPYING = 0
25         return r                          /* 返回转发指针的值, 除非对象正处于复制过程中 */
26                                          /* 尝试将复制过程中止 */
27     if CompareAndSet(&forwardingAddress(p), r, p)
28         return p                          /* 成功将复制过程中止 */
29     /* 设置失败, 意味着回收器提交成功, 或者其他赋值器线程已经将复制中止 */
30     atomic                                /* 强制重新加载当前的转发指针值 forwardingAddress(p) */
31         r ← forwardingAddress(p)
32     return r
33
34 Read(p, i):
35     p ← Access(p)
36     return p[i]
37
38 Write(p, i, value):
39     p ← Access(p)
40     p[i] ← value

```

与此同时, 赋值器线程访问对象 (包括读取或者修改对象的状态) 时需要执行如下操作:

- 1) 加载转发指针。
- 2) 如果对象的 `COPYING` 位并未设置, 则直接访问转发指针所指向的地址。
- 3) 否则, 尝试使用 `CompareAndSet` 操作来清空 `COPYING` 位 (即将其改回原本的转发指针值), 目的是中止回收器复制过程。
- 4) 如果 `CompareAndSet` 操作成功, 则使用 (`COPYING` 位已经清空的) 转发指针来作为对象指针。

5) 如果 `CompareAndSet` 操作失败, 要么是因为回收器已经成功提交了副本, 要么是由于其他赋值器线程已经中止了复制。此时赋值器线程需要使用原子读操作 (对于顺序性保障较弱的处理器才需如此) 来确保其能够读取到当前真正的转发指针值 (即由回收器或者其他赋值器线程所设置的值)。

算法 19.8 中的 `Access` 屏障辅助函数展示了上述过程, `Read` 以及 `Write` 操作均需调用该函数。

需要指出的是, 如果在 `Access` 函数中使用 `CompareAndSwap` 来替代 `CompareAndSet`, 其返回值即为转发指针的当前值, 从而节省了一次原子读操作, 如算法 19.9 所示。如果该操作成功, 则转发指针的 `COPYING` 位将被清空。

算法 19.9 Staccato 回收器：(复制过程中的) 基于 CompareAndSwap 的堆访问操作

```

1 Access(p):
2     r ← forwardingAddress(p) /* 加载转发指针 */
3     if r & COPYING = 0
4         return r /* 返回转发指针的值, 除非对象正处于复制过程中 */
5     /* 尝试将复制过程中止 */
6     r ← CompareAndSwap(&forwardingAddress(p), r, p)
7     /* 如若失败, 则意味着回收器提交成功, 或者其他赋值器线程已经将复制中止, 此时 r 的值必然正确 */
8     return r & ~COPYING /* 如果成功, 则意味着当前线程中止成功, 此时需要清空 COPYING 位 */

```

McCloskey 等 [2008] 指出, 频繁得到访问的对象将很难完成迁移, 因为其迁移过程很容易被中止。针对这一问题, 他们提出的方案是: 如果回收器探测到此类对象, 则应将其所在的页作为整理的目标页。也就是说, 回收器将增大此类对象所在页的存活对象密度, 而不是将其迁移到其他页。

另外, 如果赋值器所访问的对象恰好集中在回收器尝试移动的对象集合中, 则在短期内调用 CompareAndSwap 操作的次数将会增多, 进一步可能会导致最小赋值器使用率的下降。但这一情况通常不会发生, 因为待迁移对象通常位于存活对象密度较低的页中, 所以在同一时刻分配的、在空间上较为聚集的对象通常不存在整体迁移可能性。除此之外, 回收器也可将整理过程划分为多个阶段, 从而尽量缩短复制时间窗、降低复制过程被中止的概率。另外, 回收器在每个复制阶段中也可随机选择待整理页集合。最后, 如果同时运行多个整理线程 (不需要以同步方式运行, 但仍需满足最小赋值器使用率的要求), 则得到执行的赋值器线程数量将会减少, 相应地, 复制过程被中止的概率也会降低。

408
409

19.7.5 Chicken 回收器：在赋值器无等待前进保障条件下的尽力整理 (x86 平台)

Pizlo 等 [2008] 独立提出了一种与 Staccato 回收器类似的解决方案, 即 Chicken 回收器 (如算法 19.10 所示), 但其需要依赖 x86/x84 ~ 64 的更强的内存一致性模型 (参见表 13.1)。这意味着只有赋值器写操作才需要中止复制 (x86 平台上的读操作天然满足原子操作顺序), 且非协同握手不需要执行读屏障。Staccato 回收器和 Chicken 回收器均可以确保赋值器线程的读写操作满足无等待要求, 回收器的复制操作同样满足无等待要求, 但其可能会被赋值器线程中止。

算法 19.10 Chicken 回收器中的复制操作以及 (复制过程中的) 赋值器屏障

```

1 copyObjects(candidates):
2     for each p in candidates
3         /* 设置 COPYING 位 */
4         forwardingAddress(p) ← p | COPYING
5     waitForRaggedSynch() /* 确保所有赋值器线程均感知到 COPYING 位的变化 */
6     for each p in candidates
7         r ← allocate(length(p)) /* 分配副本 */
8         move(p, r) /* 复制数据 */
9         forwardingAddress(r) /* 副本的转发指针将指向其自身 */
10        /* 尝试“提交”副本 */
11        if not CompareAndSet(&forwardingAddress(p), p | COPYING, r)
12            /* 提交失败, 进行处理 */
13            free(r) /* 将放弃提交的副本释放 */
14            add(aborted, p) /* 记录复制失败的对象 */
15    return aborted

```

```

16
17 Read(p, i):
18     r ← forwardingAddress(p)           /* 加载转发指针 */
19     return r[i]
20
21 Write(p, i, value):
22     r ← forwardingAddress(p)           /* 加载转发指针 */
23     if r & COPYING ≠ 0                /* 返回转发指针的值, 除非对象正处于复制过程中 */
24                                     /* 尝试将复制过程中止 */
25         CompareAndSet(&forwardingAddress(p), r, r & ~COPYING)
26         /* 设置失败, 意味着回收器提交成功, 或者其他赋值器线程已经将复制中止 */
27     r ← forwardingAddress(p)          /* 加载 forwardingAddress (p) */
28     r[i] ← value

```

19.7.6 Clover 回收器：赋值器乐观无锁前进保障下的可靠整理

410

Pizlo 等还提出另一种回收器，即 Clover 回收器，该回收器能够确保赋值器访问操作在绝大多数情况下均满足无锁要求（极少数情况除外），同时也可保证回收器复制操作满足无锁要求。当赋值器和回收器之间产生竞争时，Clover 回收器会探测到这一现象并将赋值器线程阻塞，直到复制操作完成为止。Clover 回收器会将某一随机数 α 写入已完成复制的域中，并假定赋值器永远不会将 α 写入堆中。为达到这一目标，当赋值器尝试将某个域的值修改为 α 时，写屏障将捕获这一操作并将赋值器线程阻塞。

在回收器完成对象某个域的复制之后，其需要使用 CompareAndSwap 操作原子化地将该域的值设置为 α 。如果赋值器在访问该域时发现其值为 α ，则必须通过转发指针读取该域的最新值（如果该对象的副本尚未创建，则转发指针依然指向原有对象，如果已经创建，则转发指针指向新创建的副本）。即使在复制过程之前该域的值原本就是 α ，该策略依然能够正常工作。

当赋值器尝试更新某一值为 α 的域时，其必须通过转发指针来更新该域的最新位置。如果赋值器确实要将某一域的值更新为 α ，则其必须陷入阻塞，直到回收器完成该对象的复制为止（只有这样才能避免其他赋值器线程将该域的状态误认为已复制，从而读取到副本中尚未完成复制的错误数据）。算法 19.11 展示了 Clover 回收器的复制过程以及赋值器屏障的大致流程。

算法 19.11 Clover 回收器中的复制操作以及（复制过程中的）赋值器屏障

```

1 copySlot(p, i):
2     repeat
3         value ← p[i]
4         r ← forwardingAddress(p)
5         r[i] ← value
6     until CompareAndSet(&p[i], value,  $\alpha$ )
7
8 Read(p, i):
9     value ← p[i]
10    if value =  $\alpha$ 
11        r ← forwardingAddress(p)
12        value ← r[i]
13    return value
14
15 Write(p, i, newValue):
16    if newValue =  $\alpha$ 

```



```

17                                     /* 陷入阻塞，直到回收器完成复制为止 */
18  repeat
19      oldValue ← p[i]
20      if oldValue = α
21          r ← forwardingAddress(p)
22          r[i] ← newValue
23          break
24  until CompareAndSet(&src[i], oldValue, newValue)

```

对于某些类型而言，经过合理选择的 α 值能够避免与该类型的任意数值发生冲突：指针通常存在一些永远不会用到的位，而浮点数则通常不会是 NaN 值（除非程序在运行时发生浮点数计算错误）。而对于其他类型， α 值的选择应当尽量避免与程序所使用的值发生冲突。Pizlo 等 [2008] 创造性地提出了一种方案，该方案几乎可以确保 α 的值不会与程序所使用的数据发生冲突：他们使用处理器所支持的宽度最大的 CompareAndSwapWide 操作来试着一次性复制多个域。例如，现代 x86 ~ 64 处理器支持 128 位的 CompareAndSwapWide 操作，因此由 α 值造成冲突的概率便会低至 2^{-128} 。但这也意味着每个 Read/Write 操作在执行 α 值检测时必须对目标地址所在的 128 位内存进行检查。

411

19.7.7 Stopless 回收器、Chicken 回收器、Clover 回收器之间的比较

Pizlo 等 [2008] 将 Chicken 回收器、Clover 回收器与非整理式并发标记 - 清扫回收器进行了比较，除此之外他们还进一步比较了较早提出的 Stopless 回收器。定性而言，Stopless 回收器无法保证回收器的顺利执行，因为赋值器频繁更新宽副本中某一域的操作可能导致复制操作被无限期推迟。Chicken 回收器可以确保回收器的顺利执行，但其代价是某些复制操作可能会被中止。尽管 Pizlo 等声称 Clover 回收器能够确保回收器的顺利执行，但是，如果回收器不断尝试向某一赋值器频繁更新的域中写入 α 值，CompareAndSwap 操作便可能会连续写入失败。

三种回收算法均致力于赋值器堆访问操作的无锁前进保障，但彼此之间存在细微差别。Chicken 回收器能够保证赋值器的读写操作均可以达到无等待级别，而 Clover 回收器和 Stopless 回收器则只能保证写操作达到无锁级别，读操作则需要执行额外的分支。Clover 的无锁写操作只是乐观性的，因为堆写操作有可能需要等待回收器的复制操作执行完毕。

在 Clover 回收器中，对象的复制过程永远不会被中止。对于 Stopless 回收器，如果在整理阶段中两个或者多个赋值器线程在几乎相同的时间更新相同的域，则整理过程很可能被中止（详见 Pizlo 等 [2007]）。相比之下，Chicken 回收器的处理方式则稍显粗暴：任何针对正在复制的对象的写操作都会将复制过程中止。

通过基准应用程序的测试结果表明，在上述三种回收器以及非整理式并发标记 - 清扫回收器中，后者拥有最高的吞吐量（因为其读写屏障要简单得多）。用于测试的复制式回收器均需要在整理阶段开始时使用 Arnold 和 Ryder [2001] 的热拔插编译代码（hot-swapping compiled code）技术安装复制操作的专用屏障。Chicken 回收器的速度最快（复制时的执行速度会降低到原来的三分之一[⊖]），但其大量复制操作会被中止，Clover 回收器的速度次之（复制时的执行速度会降低到原来的五分之一），Stopless 回收器最慢（复制时的执行速度会降低到原来的十分之一）。所有处理器均能够在六核多处理器上表现良好。Clover 回收

⊖ 与 Pizlo 的私人交流。

器和 Stopless 回收器均会因为复制过程中吞吐量的降低而影响对实时事件的响应速度，而 Chicken 回收器的响应速度则好得多，因为其赋值器线程可以在必要情况下快速中止回收器的复制操作。

19.7.8 离散分配

上述针对实时系统整理策略的讨论表明，对于任何希望通过碎片整理来保障空间边界的实时回收器，其必须牺牲一定的吞吐量和响应速度来确保将堆的碎片化程度控制在可以接受的范围内。Chicken 回收器和 Staccato 回收器能够保证赋值器的堆访问操作达到无等待级别，但其代价是某些复制操作可能会被中止；Stopless 回收器和 Clover 回收器能够提供更强的空间保障，但其却只能保证赋值器的堆访问操作达到更弱的无锁级别。具有硬空间限制的实时回收器可能无法接受这一折中结果。

[412]

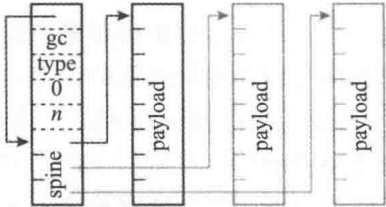
正因如此，Siebert 才一向主张将对象分配在（物理上或者逻辑上）不连续的、固定大小的内存块中，并以此限制外部碎片的总量 [Siebert, 1998, 2000, 2010]。Siebert 在其 Jamaica Java 实时系统虚拟机中使用了这一策略。Jamaica 虚拟机会将对象拆分为一组固定大小的子对象（oblet）的链表，从链表头部开始，访问下一级子对象都需要经由上一级子对象。这一布局方式将导致赋值器访问对象域所需的时间正比于该域的索引号。类似地，数组也是由多个子数组（arraylet）以二叉树方式组织而来，其最终形态将是类似于字典树（trie tree）的数据结构 [Fredkin, 1960]。因此，访问数组中某一元素所需的时间将正比于数组大小的对数。该策略的主要问题在于访问数组元素的开销会发生变化，因而其最差情况下的执行时间分析需要依赖（或者限制）赋值器能够访问的数组的静态大小。但是，Java 中数组的大小是其本身的动态属性，因而即使数组的大小可以静态预知，我们也无法证明出程序的通用空间上界。这一信息的缺失将导致树形数组的访问时间上限只能根据应用程序可能分配的最大数组来进行估算，如果连最大数组的长度也无法预估，则只能进一步根据整个堆的大小来进行估算。

针对这一问题，Pizlo 等 [2010b] 将 Metronome 回收器所用到的串联子数组（spine-based arraylet）分配策略与 Jamaica 虚拟机中的离散分配策略相结合，并称之为 Schism 回收器。由于对象和数组均会以固定大小的分片作为分配单元，所以整个系统便无需关心外部碎片问题。除此之外，对象和数组的访问均存在强时间上界：访问对象中某一域所需的间接步骤将是静态可知的（取决于域的索引号），而数组中元素的访问则需要经由串联索引（spine）来访问特定的子数组。如果进行一阶近似（且忽略高速缓存效应），则对象与数组的访问时间均为常量。Schism 回收器分配离散对象以及离散数组的策略如图 19.10 所示。对象以及数组将由堆中一个“哨兵”分片来表示。每个对象或者数组都会包含两个头部字，一个用于垃圾回收，另一个将包含类型信息。用于表示对象或者数组的哨兵分片将是这两个头部字的载体，除此之外，该分片还包括一些额外的头部字来记录其剩余结构信息。所有指向对象或者数组的引用均为指向哨兵分片的指针。

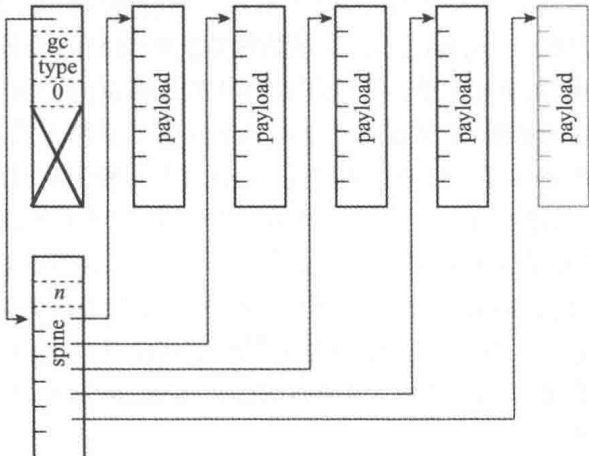
对象将以子对象链表的方式布局，如图 19.10a 所示。对于数组而言，如果其能够容纳于单个分片中，则其将以图 19.10b 的方式布局，否则，其哨兵分片将会包含一个指向串联索引的指针，而后者将包含指向每一个子数组的指针。如果串联索引足够小，则其可以“内联”到哨兵分片中，如图 19.10c 所示，否则，串联索引必须单独进行分配，如图 19.10d 所示。



- a) 由两个分片组成的对象，该对象能够容纳 6 ~ 12 个可用域。哨兵分片包含三个头部字：指向下一个分片的指针、垃圾回收专用头部字、包含类型信息的头部字。每个分片均包含指向下一个分片的指针
- b) 由单个分片构成的数组，最多可以容纳 4 个可用域。哨兵分片包含 4 个头部字：指向下一个分片的指针（为空）、垃圾回收专用头部字、包含类型信息的头部字、表示数组长度的字 ($n \leq 4$)。数组域会内联在哨兵分片中，并紧挨头部字



- c) 由多分片构成的数组，最多能够使用 3 个分片来容纳可用域（即最多 24 字节）。哨兵分片包含 5 个头部字：指向内联串联索引的指针（非空）、垃圾回收专用头部字、包含类型信息的头部字、伪长度字（为 0）、表示真正长度的字 ($4 < n \leq 24$ ，且该域在分片内的负值偏移量与 b 相同[⊖])。内联索引紧挨着头部字布局。用于容纳可用域的分片将不包含任何头部字



- d) 由多分片构成的数组，用于容纳可用域的分片数量超过 4 个（即超过 24 字节）。哨兵分片包含 4 个头部字：指向独立分配的串联索引的指针（非空）、垃圾回收专用头部字、包含类型信息的头部字、伪长度字（为 0）。该分片中的其他域将闲置不用。串联索引包含两个头部字，一个用于记录数组的真正长度，另一个用于记录转发指针[⊕]。这两个域在串联索引分片内部的偏移量均为负值。用于容纳可用域的分片将不包含任何头部字

图 19.10 Schism 回收器中的离散分配策略

Pizlo 等 [2010b], doi: 10.1145/1806596.180.6615.

© 2010 Association for Computing Machinery, Inc. 经许可后转载

Schism 回收器的创新之处在于，独立分配的数组串联索引并不需要从对象 / 数组空间进行分配。对象 / 数组的分配空间本质上是由固定大小的分片组成的集合，并使用了 Immix 标

⊖ 对于情况 b，数组的引用将与其内联数组的首元素地址相同，对于情况 c 和 d，数组的引用将与内联串联索引的首元素地址相同，因而数组长度域的偏移量均为 -1 个字。——译者注。

⊕ 转发指针应当指向哨兵分片，而数组本身的引用也为串联索引首元素地址。——译者注。

记 - 分区回收器的相关分配技术 (参见 10.3 节)。Immix 回收器中大小为 128 字节的行即为 Schism 回收器中的子对象或者子数组。Schism 回收器相当于是在 Immix 回收器中引入了离散分配以及即时并发标记技术, 同时需要使用 Dijkstra 式增量更新插入屏障。尽管内存分片永远不会移动, 但只要存在足够多的空闲分片, 任意大小的数组以及对象分配需求都可以得到满足。因此, 除了大小可变的串联索引之外, 内存碎片化将不再是一个问题。

为了限制数组串联索引所产生的碎片, Schism 回收器会在一个独立的空间中分配它们, 该空间将使用副本复制回收策略进行整理。由于数组的串联索引不会发生修改 (它们仅包含指向子数组分片的指针, 而子数组永远不会移动), 因此在回收过程中, 回收器无需担心赋值器对串联索引的更新。事实上, 对于来源空间中的串联索引主体以及其在目标空间中的副本, 赋值器可以使用其中的任意一个。另外, 每个串联索引最多只会被一个数组的哨兵分片所引用。在复制串联索引时, 回收器可以采用懒惰策略来将哨兵分片中指向串联索引主体的引用更新到其副本, 且这一操作无需与赋值器进行同步。即使回收器已经完成这一更新操作, 赋值器仍可以安全地使用来源空间中的串联索引主体, 同时也可以在下一次访问数组时从数组哨兵中获取最新副本。串联索引的副本创建完毕之后, 来源空间中的旧有主体无需任何修改便可直接丢弃, 因为此时数组哨兵分片是目标空间中新串联索引的唯一引用来源。回收器可以使用非协同握手机制来确保所有赋值器均不再访问来源空间中的串联索引。

Schism 回收器存在诸多优势。首先, 赋值器线程的堆访问操作能够满足无等待要求, 且访问时间存在严格上界 (即花费常数时间)。第二, 内存碎片问题得到严格控制: Pizlo 等 [2010b] 已经证明, 如果给定应用程序最大存活数据集中对象以及数组 (包括数组长度) 的数量与类型, 则程序所需的总内存量会严格限制在 $1.3104b$ 以内, 其中 b 为最大存活数据集的大小。第三, 与 Siebert [2000] 所提出的 Jamaica 虚拟机不同, 当堆空间足够时, Schism 回收器可以将构成数组的各个子数组在一块连续的空间内分配, 此时数组的布局方式依然如图 19.10d 所示, 唯一不同之处在于各子数组内存分片将在空间上彼此相邻。在这种情况下, 对数组元素的访问操作便可免去对串联索引的依赖。这一属性意味着与其他实时回收器相比, Schism 回收器将拥有更高的吞吐量, 而且能够在连续分配失败后降级到离散分配来确保对内存碎片情况的容忍度, 唯一的代价只是离散化数组的访问速度可能会有所降低。与单纯的并发标记 - 分区回收器 (不支持离散数组) 相比, Schism 回收器花费在读写屏障上的吞吐量开销只是前者的 77%。

如果应用程序的开发者要求数组的访问开销具有可预测性, Schism 回收器可以配置成仅使用离散分配策略来分配数组, 此时所有的数组访问操作都必须经由串联数组这一中间层。这一策略将大幅降低系统的最大停顿时间: 由于所有的分配操作都是以单个内存分片作为单元, 所以分配慢速路径的执行开销只可能是对一个 4KB 的页进行清零初始化, 对于 40MHz 的嵌入式处理器而言, 完成这一操作只需要 0.4ms。而当使用连续分配策略来分配数组时, 分配器必须首先尝试寻找一块连续内存来容纳所有分片, 在相同处理器上执行该操作可能会花费 1ms 左右的时间。

19.8 需要考虑的问题

实时系统要求对垃圾回收过程进行精细化控制以保证较短的停顿时间以及可预测的最小赋值器使用率。在本章中, 我们几乎综合运用了前面章节中的所有技术来介绍如何达到这一目标。如果不考虑并行回收以及并发回收, 则实时垃圾回收在概念上较为直接, 即如何通过

调度策略来满足良好的响应能力以及执行性能。此时我们的注意力将集中在垃圾回收算法本身，而并不过多地关注如何在应用程序中保证算法的可调度性。实时应用程序的开发者还需要对程序的最差执行时间进行精确分析，并使用这一结果进一步进行可调度性分析，从而确保程序的实时限制条件能够得到满足 [Wilhelm 等, 2008]。许多实时系统领域的文献都给出了如何对基于垃圾回收的实时应用程序进行最差时间分析以及可调度性分析 [Kim 等, 2001; Robertz and Henriksson, 2003; Chang and Wellings, 2005, 2006a, b; Chang, 2007; Cho 等, 2007, 2009; van Assche 等, 2006; Kalibera 等, 2009; Feizabadi and Back, 2005, 2007; Goh 等, 2006; Kim 等, 1999, 2000, 2001; Kim and Shin, 2004; Schoeberl, 2010; Zhao 等, 1987]。

我们将最小赋值器使用率作为衡量垃圾回收整体响应时间的主要指标，但除此之外，其他一些指标也十分重要。Printezis[2006] 指出，应用程序特定的衡量指标通常更加合适。例如，对于一个要求在固定时间窗内得到响应的周期性实时任务，只要该任务的实时响应得到满足，最小赋值器使用率便无关紧要。另外，如果赋值器线程所遭遇的垃圾回收相关停顿仅仅来自于读写屏障中的编译器内联慢速路径，或者线程本地分配缓冲区耗尽时的慢速分配路径，则最小赋值器使用率以及最大停顿时间同样无法得到保障。对于某些回收器而言 (例如 Schism 回收器)，这将是导致赋值器线程产生回收相关停顿的唯一原因 (假定回收工作在另一个处理器上执行)。我们是否应当将由此造成的停顿归咎于垃圾回收器？如果答案是肯定的，那么系统又应当如何对此做出解释？Pizlo 等 [2010a] 甚至不惜使用专用硬件来针对嵌入式处理器进行设备级别的慢速路径分析。考虑到实时应用程序的开发者通常不具备这一条件，Pizlo[2010b] 为 Schism 回收器提供了一种最差情况执行模型，该情况会强制回收器激活所有慢速路径，开发者在测试过程中可以据此对程序的最差执行时间进行合理评估。

[415]

[416]

术 语 表

完整的术语表可参见：<http://www.memorymanagement.org>。

ABA problem (ABA 问题) 无法通过 CompareAndSwap 原子操作解决的一类问题：多线程并发情况下，即使某一线程两次读取同一变量得到的值相同（设两次读取到的值都是 A），也不能断定该变量没有被修改过。因为在该线程的两次读取过程之间，可能存在另一个线程先将该变量从 A 修改到 B，再将其从 B 改回 A。

accurate (精确性) 参见类型精确性 (type-accurate)。

activation record (活动记录) 保存当前计算状态以及函数返回地址的内存结构，有时也称为帧 (frame)。

age-based collection (基于年龄的回收) 根据对象寿命将堆划分为多个空间 (space) 的回收策略。

aging space (衰老空间) 分代内部的子空间（通常位于最年轻分代内部）。对象在得到提升 (prompt) 之前必须在该空间活过数轮回收周期。

alignment (对齐) 硬件或虚拟机限制，它可能要求对象及其内部的域只能放置在特定地址界限中。

allocation (分配) 分配空闲内存单元 (free cell) 的动作。

allocator (分配器) 内存管理器中负责创建对象（但不负责将其初始化）的组件。

ambiguous pointer (模糊指针) 可能指向某一对象，也可能未指向任何对象的指针变量，参见保守式回收 (conservative collection)。

ambiguous root (模糊根) 程序根 (root) 中的模糊指针。

arraylet (子数组) 承载数组元素某个子集的固定大小的内存块 (chunk)。

barrier (屏障) 影响对象访问过程的行为（通常是由编译器引入的代码序列）。

belt (带) 带式回收器的回收增量 (increment) 集合。

best-fit allocation (最佳适应分配) 一种空闲链表分配 (free-list allocation) 策略：将对象放置在堆中能够满足分配要求且空间最小的内存单元 (cell) 中。

big bag of pages allocation, BiBoP (页簇分配) 一种分区适应分配 (segregated-fits allocation) 策略：将具有相同属性（如类型）的对象分配在同一个内存块 (block) 中，从而可以将类型信息与内存块而非单个对象相关联。

bitmap (位图) 位数组（通常是字节数组），每一位关联一个对象或者一个内存颗粒 (granule)。

bitmapped-fits allocation (位图适应分配) 使用位图来标记堆中的空闲内存颗粒的顺序适应分配 (sequential-fits allocation) 策略。

black (黑色) 如果对象是黑色的，意味着回收器已经完成对该对象的处理，并且认为它是存活的，也可参见三色抽象 (tricolour abstraction)。

black-listing (黑名单) 可能成为伪指针 (false pointer) 目标的地址区间，保守式回收可以据此减少内存泄漏 (leak)。

block (内存块) 以特定大小对齐的大块内存，其大小通常是 2 的整数次幂。

boundary tag (边界标签) 内存块边界上协助进行内存块合并 (coalescing) 的数据结构。

bounded mutator utilisation, BMU (界限赋值器使用率) 在给定时间窗或者更大时间窗内的最小赋值器使用率 (minimum mutator utilization, MMU)，与 MMU 不同，BMU 曲线是单调递增的。

breadth-first traversal (广度优先遍历) 一种对象图遍历 (traversal) 策略：在保证每个节点只访问一次的前提下，对于任意节点先辐射状地访问其所有子节点，然后再对子节点的子节点进行递归

访问。

bucket (桶) 阶 (step) 的子空间, 目的在于将对象按照年龄进行隔离。

bucket brigade (桶组) 使用桶实现分代回收 (generational collection) 的回收策略。

buddy system (伙伴系统) 一种分区适应分配策略: 以 2 的整数次幂为单位来分配内存块, 同时支持简单快速的空闲块分裂 (splitting) 以及相邻空闲块合并 (coalescing)。

buffered reference counting (缓冲引用计数) 一种引用计数 (reference counting) 形式: 赋值器先保存所有引用计数操作, 然后再将这些操作记录发送给回收器去执行。

bump pointer allocation (阶跃指针分配) 参见顺序分配 (sequential allocation)。

cache (高速缓存) 一种速度较快的存储器, 其中保存了内存中被频繁使用的数据的副本, 目的是为了提升处理器的访问速度。

cache block (高速缓存块) 参见高速缓存行 (cache line)。

cache coherence (高速缓存一致性) 判定两个或多个高速缓存中针对内存中同一数据的副本是否一致的标准。

cache hit (高速缓存命中) 高速缓存中已经存在程序所需数据的副本, 从而无需再对内存进行访问。

cache line (高速缓存行) 高速缓存和内存之间传递数据的内存单元。

cache miss (高速缓存不命中) 高速缓存中尚未包含程序所需数据的副本, 从而必须对内存进行访问。

call stack (调用栈) 线程执行过程中存储函数栈帧的动态数据结构。

car (车厢) 火车 (train) 回收器的回收单元。

car (Lisp 语言) Lisp 语言中获取 cons 单元 (cons cell) 第一个元素的操作符。

card (卡) 堆中大小为 2 的整数次幂的已对齐区域, 其空间通常较小。

card marking (卡标记) 赋值器记录回收相关指针的策略之一, 其通过写屏障来 (write barrier) 更新卡表 (card table)。

causal consistency (因果一致性) 一种一致性模型 (consistency model), 其要求如下: 如果写操作的参数需要依赖某一读操作, 则该读操作必须先于该写操作发生; 相应地, 如果某一读操作需要读取某一写操作所写入的值, 则该写操作必须先于该读操作发生。

cdr Lisp 语言中获取 cons 单元第二个元素的操作符。

cell (内存单元) 由一组连续内存颗粒组成的内存空间, 可以被分配或者释放, 甚至浪费或者不用。

Cheney scanning (Cheney 扫描) 在复制式回收 (copying collection) 中追踪 (tracing) 存活对象的一种技术, 其追踪过程无需依赖栈。

chip multiprocessor, CMP (片上多处理器) 在单个芯片中集成了多个处理器的多处理器 (multiprocessor), 参见多核 (multicore) 和众核处理器 (many-core processor)。

chunk (内存块) 由一组连续内存颗粒组成的较大内存空间。

circular first-fit allocation (环状首次适应分配) 参见循环首次适应分配 (next-fit allocation)。

coalesced reference counting (合并引用计数) 一种可以避免冗余引用计数操作的缓冲引用计数 (buffered reference counting) 策略。

coalescing (合并) 将相邻空闲内存单元组成单个空闲内存单元的操作, 也可参见分区适应分配。

coherence protocol (一致性协议) 满足特定内存一致性模型要求的高速缓存管理协议。

collection (回收) 回收器的一次运行实例, 其运行过程通常会将待回收空间中的所有死亡对象回收。

collection cycle (回收周期) 回收器的一次完整执行过程。

collector (回收器) 运行时系统中负责垃圾回收 (garbage collection) 的组件。

compacting (整理) 一种减少外部碎片 (external fragmentation) 的策略: 迁移所有得到标记的 (存活) 对象, 并且更新所有存活对象中指向已迁移对象的引用。

compaction (整理) 参见整理 (compacting)。

compaction order (整理顺序) 整理式回收器重排列对象时所遵从的顺序: 可以是任意顺序 (忽略对

象之前的排列顺序或者对象之间的关系)、线性顺序(尝试将存在引用关系的对象布置在一起)、滑动顺序(保留对象原有的排列顺序)。

completeness (完整性) 判定回收器是否能保证所有垃圾最终都能得到回收的指标;例如,引用计数算法是不完整的,因为它无法回收包含环状引用的死亡对象。

concurrent collection (并发回收) 与赋值器线程并发执行的垃圾回收过程。

condemned space (定罪空间) 待回收的内存空间或者子空间。

connectivity-based (garbage) collection, CBGC (基于相关性的(垃圾)回收) 将堆依照对象相关性划分为多个空间的回收技术。

cons cell(cons 单元) Lisp 语言中由两个元素组成的双字单元,它是将一组元素链接成表的基本单元。

conservative collection (保守式回收) 在没有编译器或者运行时系统协助的情况下的回收技术,回收器必须把栈上或者静态数据区中所有“看起来像指针”的数值当作存活对象的根。

consistency model (一致性模型) 一个内存相关规范,它规定了内存系统应当如何展现给开发者,其对读操作能从共享内存中读取到的值做出了限制。

copy reserve (复制保留区) 复制式回收中预留的内存空间。

copy collection (复制式回收) 将存活对象从一个半区(semispace)复制到另一个半区的回收策略(该过程完成后,前一个半区可以得到整体回收)。

creation space (诞生空间) 参见新生区(nursery)。

crossing map (跨越映射) 描述对象如何跨区域(通常是卡)的映射表。

dangling pointer (悬挂指针) 指向已经被内存管理器回收的对象的指针。

dead (死亡) 如果赋值器在后续执行过程中不会再访问某个对象,则可以说该对象已经死亡。

deallocation (释放) 回收已分配内存单元的操作。

deferred reference counting (延迟引用计数) 将某些引用计数操作(通常是针对局部变量)延迟的引用计数策略。

deletion barrier (删除屏障) 探测赋值器删除引用操作的写屏障,也可参见起始快照(snapshot-at-the-beginning)。

dependent load (依赖加载) 读操作的目标地址需要依赖该操作之前的另一个读操作的返回结果。

depth-first traversal (深度优先遍历) 一种图的遍历方法:在保证每个节点只访问一次的前提下,对从任意节点出发的所有分支路径深入访问到不能再深为止。

derived pointer (派生指针) 在对象引用上增加一个偏移量得到的指针。

direct collection (直接回收) 仅通过对象自身即可判定其是否存活的回收算法。

double mapping (二次映射) 将同一个物理内存页(page)以不同的保护策略映射到不同虚拟地址的技术。

double-ended queue (双端队列) 允许元素从前端(头部)添加、后端(尾部)移除的数据结构。

epoch (时段) 引用计数回收器的一个执行时间区间,回收器在该时间区间内可以避免同步操作,或者以非同步操作替代同步操作。

escape analysis (逃逸分析) 判定对象是否会脱离其诞生的方法或者线程的范围,从而成为共享对象的分析(通常是静态的)。

evacuating (迁移) 将对象从定罪空间移动到(目标空间中)新位置的操作,也可参见复制式回收或者标记-整理回收(mark-compact collection)。

explicit deallocation (显式释放) 由开发者手动执行的内存释放操作,而非由回收器自动控制。

external fragmentation (外部碎片) 内存单元之外被浪费的、无法用于分配的空间,也可参见内部碎片(internal fragmentation)。

false pointer (伪指针) 保守地假定某一值是指向某一对象的指针,但其并未真正指向任何对象,也可参见保守式回收。

false sharing (伪共享) 多个处理器同时修改落入同一个高速缓存行的数据, 导致高速缓存一致性交互增多的现象。

fast-fits allocation (快速适应分配) 一种顺序适应分配策略, 该策略通过索引查找堆中第一个或下一个满足分配要求的内存单元。

Fibonacci buddy system (斐波那契伙伴系统) 空间大小分级 (size class) 呈斐波那契序列分布的伙伴系统。

field (域) 对象中保存引用或者纯值 (scalar) 的部分。

filler object (填充对象) 为保证堆可解析性 (heap parsability) 而在真正对象之间分配的对象。

finalisation (终结) 当对象不可达 (unreachable) 时回收器所执行的动作。

finaliser (终结方法) 当回收器判定某个对象不可达时所调用的方法。

first-fit allocation (首次适应分配) 一种空闲链表分配策略, 该策略将对象放置在堆中第一个满足分配要求的内存单元中。

first-in, first-out, FIFO (先进先出) 参见队列 (queue)。

flip (翻转) 复制式回收中, 回收器在回收周期开始时将来源空间 (fromspace) 与目标空间 (tospace) 置换的操作。

floating garbage (浮动垃圾) 未能在上一个回收周期中得到回收的死亡对象。

forwarding address (转发地址) 对象得到迁移之后的新地址, 通常记录在来源空间中对象的头部 (header)。

fragmentation (碎片化) 堆中对象之间 (或者内部) 存在大量无法用于分配的小块内存, 从而导致堆使用率降低的现象, 也可参见内部碎片和外部碎片。

frame (框) 大小和对齐基址均为 2 的整数次幂的内存块。不连续内存空间中通常包含数个框。也可以参考活动记录 (activation record)。

free (空闲) 内存单元可以直接用于分配的状态。

free pointer (空闲指针) 指向内存块中空闲内存颗粒的指针, 也见顺序分配。

free-list allocation (空闲链表分配) 使用某种数据结构来记录空闲内存单元的地址和大小的顺序适应分配策略。

fromspace (来源空间) 复制式回收中, 一次回收过程开始之前对象所在的区域; 回收器将把其中的存活对象复制到目标空间。

fromspace invariant (来源空间不变式) 即“赋值器仅持有来源空间中的引用”这一不变式。

garbage (垃圾) 已经死亡但空间尚未得到回收的对象。

garbage collection, GC (垃圾回收) 当对象不再被程序使用时自动将其占用的空间回收的内存管理策略。

garbage collector (垃圾回收器) 参见回收器。

GC-check point (回收检查点) 在此处, 赋值器 (mutator) 不会主动发起回收, 但会在回收发生时将自身安全挂起的代码位置。

GC-point (回收点) 赋值器可能触发垃圾回收的代码位置 (例如对象分配相关代码)。

GC-safe point (安全回收点) 参见回收点。

generation (分代) 集中了具有某一寿命特征的对象的空间。

generational collection (分代回收) 将对象依照寿命划分为多个分代且优先回收最年轻分代的回收机制。

generational hypothesis (分代假说) 即“对象的寿命与其年龄相关”的假说; 也可参见弱分代假说 (weak generational hypothesis) 和强分代假说 (strong generational hypothesis)。

gibibyte, GiB (吉字节) 2^{30} 字节的标准计量单位。

gigabyte, GB (吉字节) 2^{30} 字节的常用计量单位。

- granule (内存颗粒)** 最小的内存分配单元, 通常为一个字或者一个双字。
- grey (灰色)** 如果回收器尚未完成对某一对象的处理, 但可以确定它是存活的, 则称该对象是灰色的; 也可参见三色抽象。
- guard page (哨兵页)** 使用禁止访问保护策略进行映射的页。
- handle (句柄)** 由运行时系统所管理的、持有对象引用的数据结构。回收器通常不会移动句柄, 但会移动其所引用的对象。
- happens-before (先于关系)** 对一组操作在内存中的发生顺序的要求。
- hard real-time system (硬实时系统)** 对响应时限有严格要求的实时系统 (real-time system); 响应超时将会导致严重的系统失败。
- header (头部)** 对象内部用于保存运行时系统所需元数据的区域。
- heap (堆)** 允许以任意顺序分配对象或者释放对象的数据结构, 在堆中分配的对象的生命期不会受到创建该对象的方法的影响。
- heap allocation (堆分配)** 从堆中分配对象的操作。
- heap parsability (堆可解析性)** 顺次逐个遍历堆中对象的能力。
- heaplet (子堆)** 堆的子集, 其所包含的对象只允许一个线程访问。
- hyperthreading (超线程)** 可参见同时多线程 (simultaneous multithreading)。
- increment (增量)** 带式回收器的回收单元, 注意不要与增量回收 (incremental collection) 混淆。
- incremental collection (增量回收)** 赋值器也需承担一小部分回收任务的回收策略; 也可参见并发回收 (concurrent collection)。
- incremental update (增量更新)** 一种解决对象丢失问题 (lost object problem) 的策略, 即赋值器将自身引发的增量更新通知给回收器。
- indirect collection (间接回收)** 一种垃圾回收策略, 即先确定所有的存活对象, 进而反推出所有的其他对象都是垃圾。
- insertion barrier (插入屏障)** 检测赋值器插入引用操作的写屏障; 也可参见增量更新。
- interior pointer (内部指针)** 指向对象内部域的派生指针。
- internal fragmentation (内部碎片)** 内存单元内部浪费的空间, 例如因向上圆整到指定大小而产生的空间浪费; 也可参见外部碎片。
- JVM (Java 虚拟机)** 执行 Java 程序的虚拟机 (virtual machine)。
- kibibyte, KiB (千字节)** 2^{10} 字节的标准计量单位。
- kilobyte, KB (千字节)** 2^{10} 字节的常用计量单位。
- large object space, LOS (大对象空间)** 为大小超过一定阈值的对象专门保留的内存空间, 该空间通常由非移动式回收器管理。
- last-in, first-out, LIFO (后进先出)** 参见栈 (stack)。
- lazy reference counting (懒惰引用计数)** 一种引用计数策略, 即只有当分配器需要内存时才将引用计数为零的对象的释放, 同时处理其子节点。
- lazy sweeping (懒惰清扫)** 仅在必要时 (通常是指需要分配新空间时) 才执行清扫。
- leak (泄漏)** 参见内存泄漏 (memory leak)。
- limit pointer (界限指针)** 指向内存块末端的指针; 也可参见顺序分配。
- linear allocation (线性分配)** 参见顺序分配。
- linearisable (线性化)** 一组以某种“看起来”不重叠的方式串行执行的并发操作的执行记录, 如果两个操作在记录中不重叠, 那么它们的发生顺序必须“看起来”与它们的调用顺序保持一致。
- linearisation point (线性化点)** 线性化记录中的操作发生瞬间的时间点。
- live (存活)** 如果赋值器在未来的执行过程中会访问某个对象, 则称该对象是存活的。
- livelock (活锁)** 在两个 (或多个) 线程 (thread) 竞争场景下, 某个 (某些) 线程永远无法向前执行的

情况。

liveness (of collector) ((回收器)存活性) 判定(并发)回收器是否最终能够完成回收过程的标准。

liveness (of object) ((对象)存活性) 判定某一对象在程序的未来执行过程中是否可能被赋值器访问的标准。

local allocation buffer, LAB (本地分配缓冲) 仅供单个线程分配空间的内存块。

locality (局部性) 程序对域或者对象的访问在时间或者空间上的集中程度;也可参见空间局部性(spatial locality)和时间局部性(temporal locality)。

lock (锁) 一种控制多个并发线程对同一资源访问的同步机制;任意时间点通常只有一个线程可以持有锁,其他线程则必须等待。

lock-free (无锁) 一种前进保障级别:尽管部分线程可能执行失败,但整个系统的执行不会被阻塞;无锁操作必然是无障碍(obstruction-free)的;也可参见非阻塞(non-blocking)。

lost object problem (对象丢失问题) 并发回收算法中可能遇到的一种问题,即赋值器和回收器的交替执行可能会导致回收器误认为某个对象不可达,进而错误地将其回收。

major collection (主回收) 分代回收中对年轻代(young generation)和年老代(old generation)都进行处理的回收。

malloc C语言标准库中从堆中分配内存的函数。

managed code (托管代码) 在托管运行时(managed run-time)上运行的程序代码。

managed run-time (托管运行时) 提供诸如自动内存管理等功能的运行时系统。

many-core processor (众核处理器) 在单个芯片上集成大量处理器的多处理器。

mark bit (标记位) 记录对象是否存活的位,可以将其保存在对象头部或者额外的标记表中。

mark-compact collection (标记-整理式回收) 一种追踪式回收(tracing collection)策略,通常分为三个或者更多阶段,回收器首先标记所有存活对象,然后通过整理来减轻碎片化。

mark-sweep collection (标记-清扫式回收) 一种追踪式回收策略,通常分为两个阶段,即首先标记所有的存活对象,然后在堆中进行清扫以回收未被标记的(死亡)对象。

mark/cons ratio (标记/构造率) 评价垃圾回收性能的一个标准,即回收器的工作量(“标记”)与赋值器的分配量(“构造”)之间的比值;也可参见cons单元。

marking (标记) 记录对象的状态为存活,一般通过设置一个标记位来实现。

mature object space, MOS (成熟对象空间) 为较老的(成熟)对象保留的内存空间,回收器通常不再关注该空间中的对象的年龄。

mebibyte, MiB (兆字节) 2^{20} 字节的标准计量单位;也可参见兆字节(megabyte)。

megabyte, MB (兆字节) 2^{20} 字节的常用计量单位;也可参见兆字节(megibyte)。

memory fence (内存屏障) 阻止处理器对某些内存访问操作进行重排列的一类同步指令。

memory leak (内存泄漏) 程序未能释放不再使用的内存空间,从而导致内存浪费的现象。

memory order (内存顺序) 对高速缓存或者内存中多个地址的读写操作顺序,以及这些操作被其他处理器察觉的顺序;也可参见程序顺序(program order)。

minimum mutator utilization, MMU (最小赋值器使用率) 给定时间窗内赋值器使用率的最小值。

minor collection (次级回收) 分代回收中只针对年轻代或者新生区的回收。

mmap 创建虚拟地址映射的UNIX系统调用。

mostly-concurrent collection (主体并发回收) 一种只需短暂停止所有赋值器线程的并发回收策略。

mostly-copying collection (主体复制式回收) 一种复制式回收策略:除了少部分被钉住(pinning)的对象之外,大部分对象都通过复制的方式进行回收。

moving collection (移动式回收) 会移动对象的回收策略。

multi-tasking virtual machine, MVM (多任务虚拟机) 可以在一次执行过程中运行多个应用程序(任务)的虚拟机。

multicore (多核) 参见片上多处理器。

multiprocessor (多处理器) 拥有超过一个处理器的计算机。

multiprogramming (多程序) 在单个处理器上执行多个进程 (processes) 或者线程 (thread)。

multitasking (多任务) 在单个处理器上执行多个任务 (tasks)。

multithreading (多线程) 在一个或多个处理器上执行多个线程。

mutator (赋值器) 即用户程序。之所以这样称呼,是因为从回收器角度来看,用户程序只是简单地改变对象图。

mutator utilisation (赋值器使用率) 赋值器在整个应用程序中所占用的 CPU 时间比例,其余 CPU 时间将被回收器占用。

nepotism (庇护) 位于非定罪空间中的死亡对象导致定罪空间中的死亡对象无法得到回收的现象。

newspace (新生空间) 用于分配新对象的内存空间。

next-fit allocation (循环首次适应分配) 一种空闲链表分配策略:将对象分配在堆中下一个满足分配要求的内存单元中。

node (节点) 参见对象。

non-blocking (非阻塞) 能够确保多个线程在竞争同一个共享资源时,任意线程都不会被永久性延迟的多线程同步模型;也可参见无障碍 (obstruction-free)、无锁 (lock-free)、无等待 (wait-free)。

non-uniform memory access (非一致内存访问) 一种多处理器架构:每个处理器都有相关联的共享内存单元,且处理器在访问自身关联的共享内存单元时速度较快。

not-marked-through, NMT (尚未标记过) 在 Pauseless 回收器中,某一引用尚未被回收器追踪到,因此不能确定其所指向的对象是否已经得到标记。

null (空) 未引用任何对象的特殊引用。

nursery (新生区) 分代式回收器中创建新对象时所使用的内存空间。

object (对象) 应用程序所使用的已分配内存单元。

object inlining (对象内联) 参见纯值替换 (scalar replacement)。

oblet (子对象) 承载对象某些域的固定大小的内存块。

obstruction-free (无障碍) 一种前进保障级别:在任意时间点,只要线程拥有足够的独占式执行时间,其便能够在有限步骤内完成操作;也可参见非阻塞。

old generation (年老年代) 对象得到提升之后所处的空间,分配器也可将对象预分配 (tenure) 到该空间。

on-stack replacement (栈上替换) 在某一函数存在调用实例的情况下替换其代码的技术。

on-the-fly collection (即时回收) 一次最多只挂起一个赋值器线程的并发回收技术。

padding (填充) 分配器为满足对齐要求而插入的额外空间。

page (页) 一个虚拟内存块。

parallel collection (并行回收) 使用多处理器或者多线程进行回收的策略;注意不要与并发回收混淆。

partial tracing (局部追踪) 仅追踪对象图的某一子集;通常是指试验删除 (trial deletion) 算法对可能是垃圾的子图进行追踪。

pause time (停顿时间) 执行万物停止式回收时赋值器被挂起的时间。

pinning (钉住) 阻止回收器移动特定对象 (通常是由于这些对象会被某些回收器无法感知到的代码访问,例如系统调用)。

pointer (指针) 对象在内存中的地址。

pointer field (指针域) 包含指针的域。

pointer reachability (指针可达性) 所有存活对象 (以及部分死亡对象) 的共同特征:存在一条从根 (root) 出发的指针链最终可以到达这些对象。

prefetching (预取) 在某个值真正得到访问之前先将其加载到高速缓存中。

- prefetching on grey (灰色预取)** 当对象被标记为灰色之后预取它的首个高速缓存行中的数据。
- pretenuring (预分配)** 分代回收中直接将对象分配到年老代的操作。
- process (进程)** 拥有独立地址空间的计算机程序运行实例；一个进程可能包含数个并发执行的线程。
- program order (程序顺序)** 程序读写多个内存地址的顺序；也可参见内存顺序。
- prolific type (富类型)** 拥有众多实例的对象类型。
- promoting (提升)** 将对象移动到年老代的行为。
- promptness (及时性)** 评判回收器是否可以在每次回收过程中都将所有垃圾全部回收的标准。
- queue (队列)** 一种先进先出数据结构，其允许从后端（尾部）添加数据、从前端（头部）移出数据。
- raw pointer (原生指针)** 即朴素指针（相对于智能指针（smart pointer）而言）。
- reachable (可达性)** 对于某一对象，是否存在一条从赋值器根出发的指针链可最终到达该对象。
- read barrier (读屏障)** 拦截赋值器加载引用操作的屏障。
- real-time (garbage) collection, RTGC (实时(垃圾)回收)** 适用于实时系统的并发回收或者增量回收。
- real-time system (实时系统)** 对事件发生后的系统响应存在时限要求的硬件或软件系统。
- reference (引用)** 用于识别对象的正规指针。
- reference count (引用计数)** 反映对象被引用次数的数值，通常位于对象的头部。
- reference counting (引用计数回收)** 通过维护每个对象被引用的次数来管理对象的垃圾回收策略。
- reference listing (引用链)** 一种垃圾回收策略：回收器为每个对象维护一个链表，链表中记录的是该对象的所有引用来源。
- region (区域)** 由开发者或者编译器管理的空间（通常由编译器自动推导出）；区域通常可以在常数时间内被清空。
- relaxed consistency (宽松一致性)** 泛指所有比顺序一致性弱的一致性模型。
- release consistency (释放一致性)** 一种一致性模型：获取（acquire）操作能够阻止后续访问操作发生在获取操作之前，但获取操作之前的访问则可以发生在获取操作之后；释放（release）操作能够阻止更早的访问操作发生在释放操作之后，但释放操作之后的操作则可以发生在释放操作之前。
- remembered set, remset (记忆集)** 记录回收器必须要处理的对象或者域的集合，在分代回收、并发回收或增量回收中，如果赋值器创建或删除的指针会影响回收器的正常工作，则赋值器需要将其记录到记忆集中。
- remset (记忆集)** 参见记忆集。
- rendezvous barrier (汇聚屏障)** 一种多线程同步模型：等待所有线程都执行到某一点之后再往下执行。
- replicate collection (副本复制回收)** 一种并发复制式回收技术，其为存活对象维护两份（甚至更多）副本。
- restricted deque (受限双端队列)** 只允许在一端添加或移除对象的双端队列。
- resurrection (复活)** 终结方法令原本不可达的对象重新可达的操作。
- root (根)** 赋值器不需要经过其他对象便可以直接访问的引用。
- root object (根对象)** 堆中直接被根所引用的对象。
- run-time system (运行时系统)** 支撑应用程序运行的代码，通常提供诸如内存管理、线程调度等服务。
- safety (安全性)** 回收器永远不得回收存活对象的特性。
- scalar (纯值)** 非引用的值。
- scalar field (纯值域)** 包含纯值的域。
- scalar replacement (纯值替换)** 一种编译器优化策略：使用局部变量来代替对象中的域。
- scanning (扫描)** 依次处理对象中的每个指针域的操作。
- scavenging (筛选)** 将存活对象从来源空间中挑拣出来的操作，也可参见复制式回收。
- schedulability analysis (可调度性分析)** 针对实时任务集合的分析，其目的在于判定是否可以在满足所有任务响应要求的前提下进行调度。

scheduler (调度器) 确定在给定时间内哪些线程应该在哪些处理器上执行的操作系统组件。

scheduling (调度) 确定何时执行一个回收单元。

segregated-fits allocation (分区适应分配) 一种内存分配策略：将对象按照空间大小分级进行划分，以达到最小碎片化的目的。

semispace (半区) 复制式回收中堆所划分成的两个空间中的一个。

semispace copying (半区复制) 参见复制式回收。

sequential allocation (顺序分配) 一种内存分配策略：从内存块一端开始连续地分配对象；通常也可称为阶跃指针分配或者线性分配。

sequential consistency (顺序一致性) 一种一致性模型，它要求所有内存访问操作看起来是逐个执行的，且每个处理器所感知到的执行顺序符合其程序顺序。

sequential fits allocation (顺序适应分配) 一种空闲链表分配策略：顺序查找空闲链表，直到找到第一个满足分配要求的内存单元为止。

sequential store buffer, SSB (顺序存储缓冲区) 一种高效的记忆集实现方式，例如槽块链。

shared pointer (共享指针) C++ 语言中支持引用计数的智能指针。

simultaneous multithreading, SMT (同时多线程) 处理器可以同时执行多个独立线程的能力。

size class (空间大小分级) 由相同分配和回收策略管理的逻辑对象集合。

slack-based scheduling (基于间隙的调度) 一种实时回收调度策略，即在没有实时任务的时候执行回收任务。

slot (槽) 参见域。

smart pointer (智能指针) 一种指针形式，其对指针的复制、解引用等操作进行重载，并据此实施内存管理操作。

snapshot-at-the-beginning (起始快照) 一种解决对象丢失问题的策略，即在回收周期开始时记录存活对象集合。

soft real-time system (软实时系统) 一类实时系统：系统必须在严格时限内对事件做出响应以确保服务质量；如果响应超出时限，则服务质量将受到影响。

space (空间) 由某种回收算法所管理的堆的子集。

spatial locality (空间局部性) 如果程序访问某个存储器地址后，又在较短时间内访问邻近的存储器地址，则程序具有良好的空间局部性；两次访问的地址越接近（例如同一页或者同一个高速缓存行），空间局部性越好。

spin lock (自旋锁) 一种同步锁模型：线程在尝试获取锁时，如果获取失败（即锁被其他线程占有），则简单地执行循环，直到获取锁为止。

splitting (分裂) 将一个内存单元拆分成两个相邻的内存单元；也可参见分区适应分配。

stack (栈) 只允许在前端（上方）添加和移出元素的后进先出数据结构；也可参见调用栈。

stack allocation (栈上分配) 将对象直接分配在当前方法的栈帧上。

stack barrier (栈屏障) 用于拦截线程返回（或抛出异常）操作越过调用栈中指定栈帧的屏障。

stack frame (栈帧) 在调用栈上分配的活动记录。

stack map (栈映射) 记录调用栈中哪些地址可能存在指针的数据结构。

static allocation (静态分配) 在编译期就可以确定对象地址的内存分配。

step (阶) 分代内部依照对象年龄进行隔离的子空间。

sticky reference count (粘性引用计数) 已经达到上限的引用计数值，该值将不受后续指针更新操作的影响而变化。

stop-the-world collection (万物静止式回收) 需要在垃圾回收过程中挂起所有赋值器线程的回收策略。

store buffer (存储缓冲区) 参见写缓冲区。

strict consistency (严格一致性) 一种一致性模型，它要求所有内存访问操作及原子操作都能以相同

- 的顺序被所有处理器感知到。
- strong generational hypothesis (强分代假说) 即“越老的对象越不容易死亡”这一假说。
- strong reference (强引用) 会影响对象可达性的引用;一般的引用都是强引用。
- strong tricolour invariant (强三色不变式) 一种三色抽象不变式,它要求黑色对象不得引用白色对象。
- sweeping (清扫) 在堆或堆的子集中进行线性扫描,并将未标记的(死亡)对象回收的过程。
- symmetric multiprocessor (对称多处理器) 共享内存单元与处理器相互独立的多处理器。
- task (任务) 进程或者线程的工作单元,通常用于实时系统。
- tebabyte, TiB (太字节) 2^{40} 字节的标准计量单位;也可参见太字节(terabyte)。
- temporal locality (时间局部性) 如果被访问过的存储器地址在较短时间内被再次访问,则程序具有良好的时间局部性;同一个地址的两次访问间隔时间越短,时间局部性越好。
- tenuring (提升) 参见提升。
- terabyte (太字节) 2^{40} 字节的常用计量单位;也可参见太字节(tebabyte)。
- test-and-set lock (检测并设置锁) 参见自旋锁。
- test-and-test-and-set lock (检测-检测并设置锁) 一种开销较低的检测并设置锁,即程序仅在锁“看起来”未被占用时才使用昂贵的硬件原子操作。
- thread (线程) 某一地址空间内的一个顺序执行路径,是操作系统的最小调度单元;也可参见进程。
- threaded compaction (引线整理) 一种整理技术:将引用了某一对象的所有对象链接起来,从而可以找到引用了该对象的所有对象。
- tidy pointer (正规指针) 可以用作对象引用的正规指针。
- time-based scheduling (基于时间的调度) 一种实时回收调度策略:为回收器预留出特定比例的独占式执行时间,且在回收器执行期间赋值器将处于挂起状态。
- tospace (目标空间) 复制式回收中用于容纳存活对象的半区。
- tospace invariant (目标空间不变式) 即“赋值器仅持有指向目标空间对象的引用”这一不变式。
- tracing (追踪) 对整个或者部分对象图进行访问以找到可达对象的过程。
- tracing collection (追踪式回收) 一种间接回收技术:以追踪的方式获取对象图中的所有存活对象,进而反推出所有垃圾对象。
- train (列车) 成熟对象空间回收器的一个组件。
- transaction (事务) 必须原子性地执行的一组读写操作集合。
- transaction abort (事务中止) 事务不成功地结束,其造成的所有影响都被抛弃。
- transaction commit (事务提交) 事务成功地结束,且其作用可见。
- translation lookaside buffer, TLB (转译后备缓冲区) 用于缓存(部分)虚拟地址和物理地址映射关系的内存管理单元。
- traversal (遍历) 访问图中的每个节点,且确保每个节点仅被访问一次。
- trial deletion (试验删除) 临时性地删除一个引用,进而确定该操作是否会导致某一对象的引用计数变为零。
- tricolour abstraction (三色抽象) 一种对垃圾回收工作方式的描述:将对象划分为白色(尚未访问到)、黑色(已经访问过)、灰色(剩余工作,后续还会访问到)。
- type-accurate (类型精确) 即回收器可以精确地识别出每个包含指针的槽或者根。
- ulterior reference counting (超引用计数) 一种引用计数策略:使用复制的方式管理年轻对象,使用引用计数方法管理年老对象。
- unique pointer (唯一指针) 一种智能指针形式,它可以确保目标对象仅被该指针所引用。
- virtual machine, VM (虚拟机) 对底层硬件细节和操作系统进行抽象的运行时系统。
- wait-free (无等待) 一种多线程同步模型:无论是针对整个系统,还是针对每个线程,线程操作都可以在有限步骤内完成;也可参见非阻塞。

wavefront (回收波面) 垃圾回收过程中由灰色对象(即正在处理的对象)组成的边界,它将黑色对象(已经处理过)和白色对象(尚未处理过)隔离。

weak consistency (弱一致性) 将每种原子操作都当作完全内存屏障的一致性模型。

weak generational hypothesis (弱分代假说) 即“大多数对象都在年轻时死亡”的假说。

weak reference (弱引用) 不会对目标对象的可达性造成影响的引用;Java 提供多种方式的弱引用。

weak tricolour invariant (弱三色不变式) 一种三色抽象不变式,它要求从黑色对象可达的白色对象必须从灰色对象可达(可能是直接可达,也可能经过一条白色对象链)。

white (白色) 如果一个对象尚未被回收器处理,则称该对象是白色的;回收周期完成后,白色对象即为死亡对象;也可参见三色抽象。

wilderness (拓展块) 堆中最后一个空闲内存块。

wilderness preservation (拓展块保护) 一种内存分配策略:将拓展块作为内存分配的最后备选空间。

work stealing (工作窃取) 一种线程间负载均衡策略:轻负载线程从重负载线程中拉取部分工作。

work-based scheduling (基于工作的调度) 一种实时回收调度策略,它要求每个赋值器工作单元都必须处理一定的回收工作。

worst-case execution time, WCET (最差执行时间) 某一硬件平台下完成某一操作所需的最长时间;该值对于硬实时系统的可调度性分析不可或缺。

write barrier (写屏障) 用于拦截赋值器存储引用操作的屏障。

write buffer (写缓冲区) 存放待写入内存的数据的缓冲区。

young generation (年轻代) 参见新生区。

zero count table, ZCT (零引用表) 记录引用计数为零的对象的表。

参考文献

本书的参考文献包含超过 400 条引用，而位于我们网站 (<http://www.cs.kent.ac.uk/~rej/gcbib/>) 的综合数据库包含了超过 2500 条垃圾回收相关的文献，该数据库支持在线搜索，同时还支持 BibTEX、PostScript、PDF 格式的下载。除了相关文章、论文、书籍之外，该参考文献还包括了某些文献的摘要，对于大多数存在电子版的文献，我们还给出了其 URL 以及 DOI 信息。我们将持续更新本书参考文献，并将其作为一项社区服务。如果你有更多文献（或者修正意见），欢迎联系 Richard (R.E.Jones@kent.ac.uk)，我们将不胜感激。

Santosh Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *International Conference on Parallel Processing*, University Park, PA, August 1987, pages 243–246. Pennsylvania State University Press. Also technical report CSRD 620, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development. xvii, 316, 317, 318, 326, 466

Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA 2004*, pages 224–236. doi: 10.1145/1028976.1028995. xx, 32, 38, 46, 301, 302, 319

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. WRL Research Report 95/7, Digital Western Research Laboratory, September 1995. 237

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996. doi: 10.1109/2.546611. 237

Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, 1998. 188, 189

Rafael Alonso and Andrew W. Appel. An advisor for flexible working sets. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, pages 153–162. ACM Press. doi: 10.1145/98457.98753. 208, 209

Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987. doi: 10.1016/0020-0190(87)90175-X. 125, 171

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989a. doi: 10.1002/spe.4380190206. 121, 122, 125, 195, 197

Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2: 153–162, 1989b. doi: 10.1007/BF01811537. 171, 172

Andrew W. Appel. Tutorial: Compilers and runtime systems for languages with garbage collection. In *PLDI 1992*. doi: 10.1145/143095. 113

Andrew W. Appel. Emulating write-allocate on a no-write-allocate cache. Technical Report TR-459-94, Department of Computer Science, Princeton University, June 1994. 100, 165, 166

- Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Department of Computer Science, Princeton University, March 1994. 171
- Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47-74, January 1996. doi: 10.1017/S095679680000157X. 171
- Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In PLDI 1988, pages 11-20. doi: 10.1145/53990.53992. xvii, 316, 317, 318, 340, 352, 467
- J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996. 146
- Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In PLDI 2001, pages 168-179. doi: 10.1145/378795.378832. 412
- Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998, pages 119-129. ACM Press. doi: 10.1145/277651.277678. 267, 268, 280
- Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, Atlanta, GA, 2008, pages 245-254. ACM Press. doi: 10.1145/1450058.1450092. 400
- Thomas H. Axford. Reference counting of cyclic graphs for functional programs. *Computer Journal*, 33(5):466-470, 1990. doi: 10.1093/comjnl/33.5.466. 67
- Alain Azagury, Elliot K. Kolodner, and Erez Petrank. A note on the implementation of replication-based garbage collection for multithreaded applications and multiprocessor environments. *Parallel Processing Letters*, 9(3):391-399, 1999. doi: 10.1142/S0129626499000360. 342
- Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *12th International Conference on Compiler Construction*, Warsaw, Poland, May 2003, pages 185-199. Volume 2622 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-36579-6_14. 369
- Hezi Azatchi, Yossi Levononi, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA 2003*, pages 269-281. doi: 10.1145/949305.949329. 331
- Azul. Pauseless garbage collection. White paper AWP-005-020, Azul Systems Inc., July 2008. 355, 361
- Azul. Comparison of virtual memory manipulation metrics. White paper, Azul Systems Inc., 2010. 360
- David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *15th European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001, pages 207-235. Volume 2072 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-45337-7_12. 59, 67, 72, 108, 157, 366, 373
- David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI 2001, pages 92-103. doi: 10.1145/378795.378819. 67, 366

- David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL 2003*, pages 285–298. doi: 10.1145/604131.604155. 323, 391, 394, 395
- David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *LCTES 2003*, pages 81–92. doi: 10.1145/780732.780744. 404
- David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *OOPSLA 2004*, pages 50–68. doi: 10.1145/1035292.1028982. 77, 80, 134
- David F. Bacon, Perry Cheng, David Grove, and Martin T. Vechev. Syncopation: Generational real-time garbage collection in the Metronome. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, Chicago, IL, June 2005, pages 183–192. *ACM SIGPLAN Notices* 40(7), ACM Press. doi: 10.1145/1065910.1065937a. 399
- Scott B. Baden. Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 331–342. Addison-Wesley, 1983. 61, 63
- Brenda Baker, E. G. Coffman, and D. E. Willard. Algorithms for resolving conflicts in dynamic storage allocation. *Journal of the ACM*, 32(2):327–343, April 1985. doi: 10.1145/3149.335126. 139
- Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. doi: 10.1145/359460.359470. Also AI Laboratory Working Paper 139, 1977. xvii, 138, 277, 316, 317, 318, 337, 340, 341, 342, 347, 361, 377, 384, 385
- Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992a. doi: 10.1145/130854.130862. 104, 139, 361
- Henry G. Baker. CONS should not CONS its arguments, or a lazy alloc is a smart alloc. *ACM SIGPLAN Notices*, 27(3), March 1992b. doi: 10.1145/130854.130858. 147
- Henry G. Baker. ‘Infant mortality’ and generational garbage collection. *ACM SIGPLAN Notices*, 28(4):55–57, April 1993. doi: 10.1145/152739.152747. 106
- Jason Baker, Antonio Cuneì, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *International Conference on Compiler Construction*, Braga, Portugal, March 2007. Volume 4420 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-71229-9_5. 432
- Jason Baker, Antonio Cuneì, Tomas Kalibera, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 21(12):1572–1606, 2009. doi: 10.1002/cpe.1391. Supersedes Baker et al [2007]. 171
- Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA 2003*, pages 255–268. doi: 10.1145/949305.949328. 319, 320
- Katherine Barabash, Ori Ben-Yitzhak, Irit Gofit, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, November 2005. doi: 10.1145/1108970.1108972. 284, 319, 320, 474
- David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening

- boundary. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995, pages 301–314. ACM SIGPLAN Notices 30(6), ACM Press. doi: 10.1145/207110.207164. 123
- David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI 1993*, pages 187–196. doi: 10.1145/155090.155108. 114
- Joel F. Bartlett. Compacting garbage collection with ambiguous roots. WRL Research Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988a. Also appears as Bartlett [1988b]. 30, 104
- Joel F. Bartlett. Compacting garbage collection with ambiguous roots. *Lisp Pointers*, 1(6): 3–12, April 1988b. doi: 10.1145/1317224.1317225. 432
- Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical Note TN-12, DEC Western Research Laboratory, Palo Alto, CA, October 1989a. 170, 192
- Joel F. Bartlett. SCHEME->C: a portable Scheme-to-C compiler. WRL Research Report 89/1, DEC Western Research Laboratory, Palo Alto, CA, January 1989b. 170
- George Belotsky. C++ memory management: From fear to triumph. O'Reilly linuxdevcenter.com, July 2003. 3
- Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984. doi: 10.1145/579.587. 309
- Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000, pages 117–128. ACM SIGPLAN Notices 35(11), ACM Press. doi: 10.1145/356989.357000. 102
- Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. doi: 1721.1/16428. Technical report MIT/LCS/TR-178. 103, 140
- Stephen Blackburn and Kathryn S. McKinley. Immix garbage collection: Mutator locality, fast collection, and space efficiency. In *PLDI 2008*, pages 22–32. doi: 10.1145/1375581.1375586. 30, 100, 152, 153, 154, 159, 186
- Stephen Blackburn, Robin Garner, Chris Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiederman. The DaCapo benchmarks: Java benchmarking development and analysis (extended version). Technical report, The DaCapo Group, 2006a. 59, 114, 125
- Stephen Blackburn, Robin Garner, Kathryn S. McKinley, Amer Diwan, Samuel Z. Guyer, Antony Hosking, J. Eliot B. Moss, Darko Stefanović, *et al.* The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, October 2006b, pages 169–190. ACM SIGPLAN Notices 41(10), ACM Press. doi: 10.1145/1167473.1167488. 10
- Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In *ISMM 2004*, pages 143–151. doi: 10.1145/1029873.1029891. 202, 203

- Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In ISMM 2002, pages 175–184. doi: 10.1145/512429.512452. 80
- Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA 2003, pages 344–458. doi: 10.1145/949305.949336. 49, 55, 60, 61, 108, 157, 158, 159, 322
- Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In OOPSLA 2001, pages 342–352. doi: 10.1145/504282.504307. 110, 132
- Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI 2002, pages 153–164. doi: 10.1145/512529.512548. 130, 131, 140, 202
- Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2004a, pages 25–36. ACM SIGMETRICS Performance Evaluation Review 32(1), ACM Press. doi: 10.1145/1005686.1005693. 46, 49, 54, 55, 79, 88, 105, 126, 130, 203
- Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *26th International Conference on Software Engineering*, Edinburgh, May 2004b, pages 137–146. IEEE Computer Society Press. doi: 10.1109/ICSE.2004.1317436. 26, 107, 116, 195, 196
- Stephen M. Blackburn, Matthew Hertz, Kathryn S. McKinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, 29(1):1–57, 2007. doi: 10.1145/1180475.1180477. 110, 132
- Bruno Blanchet. Escape analysis for object oriented languages: Application to Java. In OOPSLA 1999, pages 20–34. doi: 10.1145/320384.320387. 147
- Ricki Blau. Paging on an object-oriented personal computer for Smalltalk. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Minneapolis, MN, August 1983, pages 44–54. ACM Press. doi: 10.1145/800040.801394. Also appears as Technical Report UCB/CSD 83/125, University of California, Berkeley, Computer Science Division (EECS). 50
- Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In PLDI 1999, pages 104–117. doi: 10.1145/301618.301648. xviii, xx, 256, 289, 292, 377, 378, 379, 380, 381, 382, 383, 384, 385, 391, 404, 406, 468, 477
- Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980. doi: 10.1145/357103.357104. 67
- Hans-Juergen Boehm. Mark-sweep vs. copying collection and asymptotic complexity. http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html, September 1995. 26
- Hans-Juergen Boehm. Reducing garbage collector cache misses. In ISMM 2000, pages 59–64. doi: 10.1145/362422.362438. 23, 27
- Hans-Juergen Boehm. Destructors, finalizers, and synchronization. In POPL 2003, pages 262–272. doi: 10.1145/604131.604153. 218, 219, 221
- Hans-Juergen Boehm. The space cost of lazy reference counting. In *31st Annual ACM Symposium on Principles of Programming Languages*, Venice, Italy, January 2004, pages 210–219. ACM SIGPLAN Notices 39(1), ACM Press. doi: 10.1145/604131.604153. 59, 60

- Hans-Juergen Boehm. Space efficient conservative garbage collection. In PLDI 1993, pages 197–206. doi: 10.1145/155090.155109. 105, 168
- Hans-Juergen Boehm and Mike Spertus. Garbage collection in the next C++ standard. In ISMM 2009, pages 30–38. doi: 10.1145/1542431.1542437. 3, 4
- Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988. doi: 10.1002/spe.4380180902. 22, 23, 31, 79, 94, 95, 96, 104, 137, 163, 166, 209, 280, 346
- Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In PLDI 1991 [PLDI 1991], pages 157–164. doi: 10.1145/113445.113459. xvii, 202, 315, 316, 318, 323, 326, 466
- Michael Bond and Kathryn McKinley. Tolerating memory leaks. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, October 2008, pages 109–126. ACM SIGPLAN Notices 43(10), ACM Press. doi: 10.1145/1449764.1449774. 208
- Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In OOPSLA 2001, pages 353–366. doi: 10.1145/504282.504308. 209
- Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Transactions on Programming Languages and Systems*, 28(5):908–941, September 2006. doi: 10.1145/1152649.1152652. 209
- R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989. doi: 10.1145/65979.65981. 139
- Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In LFP 1984, pages 256–262. doi: 10.1145/800055.802042. 340, 341, 347, 361, 386, 404, 405
- David R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Conference on Functional Programming and Computer Architecture*, Nancy, France, September 1985, pages 273–288. Volume 201 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-15975-4_42. 67
- F. Warren Burton. A buddy system variation for disk storage allocation. *Communications of the ACM*, 19(7):416–417, July 1976. doi: 10.1145/360248.360259. 96
- Albin M. Butters. Total cost of ownership: A comparison of C/C++ and Java. Technical report, Evans Data Corporation, June 2007. 1, 4
- Brad Calder, Chandra Krintz, S. John, and T. Austin. Cache-conscious data placement. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998, pages 139–149. ACM SIGPLAN Notices 33(11), ACM Press. doi: 10.1145/291069.291036. 50
- D. C. Cann and Rod R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS-88-129, Department of Computer Science, Colorado State University, Fort Collins, CO, 1988. 61
- Dante Cannarozzi, Michael P. Plezbert, and Ron Cytron. Contaminated garbage collection. In PLDI 2000, pages 264–273. doi: 10.1145/349299.349334. 147
- Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg

- Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992. doi: 10.1145/142137.142141. 340
- Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, November 1986, pages 119–130. ACM SIGPLAN Notices 21(11), ACM Press. doi: 10.1145/28697.28709. 114, 138
- CC 2005. *14th International Conference on Compiler Construction*, Edinburgh, April 2005. Volume 3443 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/b107108. 451
- Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing. In *26th Annual Symposium on Foundations of Computer Science*, Portland, OR, October 1985, pages 261–288. IEEE Computer Society Press. doi: 10.1109/SFCS.1985.48. 195
- Yang Chang. *Garbage Collection for Flexible Hard Real-time Systems*. PhD thesis, University of York, 2007. 415
- Yang Chang and Andy Wellings. Integrating hybrid garbage collection with dual priority scheduling. In *11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2005, pages 185–188. IEEE Press, IEEE Computer Society Press. doi: 10.1109/RTCSA.2005.56. 415
- Yang Chang and Andy Wellings. Low memory overhead real-time garbage collection for Java. In *4th International Workshop on Java Technologies for Real-time and Embedded Systems*, Paris, France, October 2006a. doi: 10.1145/1167999.1168014. 415
- Yang Chang and Andy J. Wellings. Hard real-time hybrid garbage collection with low memory requirements. In *27th IEEE Real-Time Systems Symposium*, December 2006b, pages 77–86. doi: 10.1109/RTSS.2006.25. 415
- David R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, August 1987. doi: 1911/16127. 104
- David R. Chase. Safety considerations for storage allocation optimizations. In *PLDI 1988*, pages 1–10. doi: 10.1145/53990.53991. 104
- A. M. Cheadle, A. J. Field, and J. Nyström-Persson. A method specialisation and virtualised execution environment for Java. In David Gregg, Vikram Adve, and Brian Bershad, editors, *4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, WA, March 2008, pages 51–60. ACM Press. doi: 10.1145/1346256.1346264. 170, 341
- Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton Jones, and R.L. While. Non-stop Haskell. In *5th ACM SIGPLAN International Conference on Functional Programming*, Montreal, September 2000, pages 257–267. ACM Press. doi: 10.1145/351240.351265. 170, 171
- Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton Jones, and Lyndon While. Exploring the barrier to entry — incremental generational garbage collection for Haskell. In *ISMM 2004*, pages 163–174. doi: 10.1145/1029873.1029893. 99, 170, 171, 340, 341
- Wen-Ke Chen, Sanjay Bhansali, Trishul M. Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI 2006*, pages 332–340. doi: 10.1145/1133981.1134021. 53
- C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13

- (11):677–8, November 1970. doi: 10.1145/362790.362798. 43, 44
- Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI 2001*, pages 125–136. doi: 10.1145/378795.378823. 7, 187, 289, 290, 304, 377, 382, 384, 468
- Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998, pages 162–173. ACM SIGPLAN Notices 33(5), ACM Press. doi: 10.1145/277650.277718. 110, 132, 146
- Perry Sze-Din Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, September 2001. SCS Technical Report CMU-CS-01-174. 289, 382, 468
- Chen-Yong Cher, Antony L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In Shubh Mukherjee and Kathryn S. McKinley, editors, *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004, pages 199–210. ACM SIGPLAN Notices 39(11), ACM Press. doi: 10.1145/1024393.1024417. 27, 51
- Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM 1998*, pages 37–48. doi: 10.1145/301589.286865. 53
- Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI 1999*, pages 1–12. doi: 10.1145/301618.301633. 50
- Hyeonjoong Cho, Chewoo Na, Binoy Ravindran, and E. Douglas Jensen. On scheduling garbage collector in dynamic real-time systems with statistical timing assurances. *Real-Time Systems*, 36(1–2):23–46, 2007. doi: 10.1007/s11241-006-9011-0. 415
- Hyeonjoong Cho, Binoy Ravindran, and Chewoo Na. Garbage collector scheduling in dynamic, multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):845–856, June 2009. doi: 10.1109/TPDS.2009.20. 415
- Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD’s advanced synchronization facility within a complete transactional memory stack. In *European Conference on Computer Systems (EuroSys)*, 2010, pages 27–40. ACM Press. doi: 10.1145/1755913.1755918. 271
- Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, February 1977. doi: 10.1145/359423.359427. 73
- Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In *VEE 2005*, pages 46–56. doi: 10.1145/1064979.1064988. 355, 361
- Marshall P. Cline and Greg A. Lomow. *C++ FAQs: Frequently Asked Questions*. Addison-Wesley, 1995. 3
- William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. In *PLDI 1997*, pages 97–108. doi: 10.1145/258915.258925. 128
- Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4): 532–553, 1983. doi: 10.1145/69575.357226. 34

- George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960. doi: 10.1145/367487.367501. xxiii, 2, 57
- W. T. Comfort. Multiword list items. *Communications of the ACM*, 7(6):357–362, June 1964. doi: 10.1145/512274.512288. 94
- Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *LFP 1992*, pages 43–52. doi: 10.1145/141471.141501. 209
- Erik Corry. Optimistic stack allocation for Java-like languages. In *ISMM 2006*, pages 162–173. doi: 10.1145/1133956.1133978. 147
- Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988. doi: 10.1007/BF01407816. 299, 300
- David Detlefs. Automatic inference of reference-count invariants. In *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, Venice, Italy, January 2004a. 152
- David Detlefs. A hard look at hard real-time garbage collection. In *7th International Symposium on Object-Oriented Real-Time Distributed Computing*, Vienna, May 2004b, pages 23–32. IEEE Press. doi: 10.1109/ISORC.2004.1300325. Invited paper. 377, 385
- David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *2nd Java Virtual Machine Research and Technology Symposium*, San Francisco, CA, August 2002a. *USENIX*. 196, 197, 199, 201, 319
- David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM 2004*, pages 37–48. doi: 10.1145/1029873.1029879. 150, 159
- David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie Mellon University, Pittsburgh, PA, May 1990. 340
- David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. In *20th ACM Symposium on Distributed Computing*, Newport, Rhode Island, August 2001, pages 190–199. ACM Press. doi: 10.1145/383962.384016. 365
- David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. *Distributed Computing*, 15:255–271, 2002b. doi: 10.1007/s00446-002-0079-z. 365
- John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990. 338, 340, 366
- L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976. doi: 10.1145/360336.360345. 61, 105
- Sylvia Dieckmann and Urs Hölzle. The allocation behaviour of the SPECjvm98 Java benchmarks. In Rudolf Eigenman, editor, *Performance Evaluation and Benchmarking with Realistic Applications*, chapter 3, pages 77–108. MIT Press, 2001. 59
- Sylvia Dieckmann and Urs Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In Rachid Guerraoui, editor, *13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, July 1999, pages 92–115. Volume 1628 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-48743-3_5. 59, 114, 125
- Edgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens.

- On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces: International Summer School*, volume 46 of *Lecture Notes in Computer Science*, pages 43–56. Springer-Verlag, Marktoberdorf, Germany, 1976.
doi: 10.1007/3-540-07994-7_48. 12, 20, 309, 315, 316, 323, 466, 467
- Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978. doi: 10.1145/359642.359655. xvii, 12, 20, 309, 315, 316, 317, 318, 323, 330
- Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.
doi: 10.1147/sj.391.0151. 30, 100, 101, 150, 151, 152, 186
- Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *PLDI 1992*, pages 273–282.
doi: 10.1145/143095.143140. 179, 180, 183, 184
- Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs using copying garbage collection. In *POPL 1994*, pages 1–14.
doi: 10.1145/174675.174710. 100, 165
- Julian Dolby. Automatic inline allocation of objects. In *PLDI 1997*, pages 7–17.
doi: 10.1145/258915.258918. 148
- Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI 2000*, pages 345–357. doi: 10.1145/349299.349344. 148
- Julian Dolby and Andrew A. Chien. An evaluation of automatic object inline allocation techniques. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 1998, pages 1–20. ACM SIGPLAN Notices 33(10), ACM Press. doi: 10.1145/286936.286943. 148
- Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL 1994*, pages 70–83. doi: 10.1145/174675.174673. xxi, 107, 108, 329, 331, 369
- Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993, pages 113–123. ACM Press.
doi: 10.1145/158511.158611. 107, 108, 146, 329
- Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *PLDI 2000*, pages 274–284. doi: 10.1145/349299.349336. 330, 331
- Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In *ISMM 2002*, pages 76–87.
doi: 10.1145/512429.512439. 109, 110, 146
- Kevin Donnelly, Joe Hallett, and Assaf Kfoury. Formal semantics of weak references. In *ISMM 2006*, pages 126–137. doi: 10.1145/1133956.1133974. 228
- R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *PLDI 1993*, pages 207–216. doi: 10.1145/155090.155110. 220
- ECOOP 2007, Erik Ernst, editor. *21st European Conference on Object-Oriented Programming*, Berlin, Germany, July 2007. Volume 4609 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-73589-2. 440, 460

- Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference*, Portland, OR, August 1992. USENIX. 59, 74
- Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *ACM/IEEE Conference on Supercomputing*, San Jose, CA, November 1997. doi: 10.1109/SC.1997.10059. xvii, 249, 277, 280, 281, 283, 289, 299, 304, 474
- A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8): 3–6, August 1958. doi: 10.1145/368892.368907. 169
- Shahrooz Feizabadi and Godmar Back. Java garbage collection scheduling in utility accrual scheduling environments. In *3rd International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, San Diego, CA, 2005. 415
- Shahrooz Feizabadi and Godmar Back. Garbage collection-aware sheduling utility accrual scheduling environments. *Real-Time Systems*, 36(1–2), July 2007. doi: 10.1007/s11241-007-9020-7. 415
- Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969. doi: 10.1145/363269.363280. 43, 44, 50, 107
- Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *1st International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, March 2003, pages 241–252. IEEE Computer Society Press. doi: 10.1109/CGO.2003.1191549. 190
- David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters*, 3(1):29–32, July 1974. doi: 10.1016/0020-0190(74)90044-1. 36
- Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In *ISMM 2000*, pages 111–120. doi: 10.1145/362422.362472. 5, 77, 138, 202
- Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM 2001*. xvii, xx, 34, 36, 248, 278, 280, 282, 283, 284, 288, 289, 292, 298, 300, 301, 303, 304, 357, 465, 468, 474
- John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, Berkeley, CA, 1981, pages 14–19. ACM Press. doi: 10.1145/800206.806364. 113
- John K. Foderaro, Keith Sklower, Kevin Layer, et al. *Franz Lisp Reference Manual*. Franz Inc., 1985. 27
- Daniel Frampton, David F. Bacon, Perry Cheng, and David Grove. Generational real-time garbage collection: A three-part invention for young objects. In *ECOOP 2007*, pages 101–125. doi: 10.1007/978-3-540-73589-2_6. 399
- Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960. doi: 10.1145/367390.367400. 413
- Daniel P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1):41–45, January 1979. doi: 10.1016/0020-0190(79)90091-7. 66, 67
- Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In *ISMM 2007*, pages 43–54. doi: 10.1145/1296907.1296915. 23, 26, 28, 29

- Alex Garthwaite. *Making the Trains Run On Time*. PhD thesis, University of Pennsylvania, 2005. 194
- Alex Garthwaite, Dave Dice, and Derek White. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification. In *VEE 2005*, pages 24–34. doi: 10.1145/1064979.1064985. 101, 195
- Alexander T. Garthwaite, David L. Detlefs, Antonios Printezis, and Y. Srinivas Ramakrishna. Method and mechanism for finding references in a card in time linear in the size of the card in a garbage-collected heap. United States Patent 7,136,887 B2, Sun Microsystems, November 2006. xxi, 200, 201
- David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction*, Berlin, April 2000, pages 82–93. Volume 2027 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-46423-9_6. 147, 148
- GC 1990, Eric Jul and Niels-Christian Juul, editors. *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, October 1990. 444, 459
- GC 1991, Paul R. Wilson and Barry Hayes, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. 444, 460
- GC 1993, Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors. *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, October 1993. 444, 460
- Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Montréal, Canada, October 2007, pages 57–76. ACM SIGPLAN Notices 42(10), ACM Press. doi: 10.1145/1297027.1297033. 10
- Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In *OOPSLA 2005*, pages 97–116. doi: 10.1145/1094811.1094819. 132
- O. Goh, Yann-Hang Lee, Z. Kaakani, and E. Rachlin. Integrated scheduling with garbage collection for real-time embedded applications in CLI. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, Gyeongju, Korea, April 2006. IEEE Press. doi: 10.1109/ISORC.2006.41. 415
- Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *PLDI 1991 [PLDI 1991]*, pages 165–176. doi: 10.1145/113445.113460. 171, 172
- Benjamin Goldberg. Incremental garbage collection without tags. In *European Symposium on Programming*, Rennes, France, February 1992, pages 200–218. Volume 582 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-55253-7_12. 171
- Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *LFP 1992*, pages 53–65. doi: 10.1145/141471.141504. 171, 172
- Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Conference on Functional Programming and Computer Architecture*, La Jolla, CA, June 1995, pages 293–305. ACM Press. doi: 10.1145/224164.224219. 165
- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition edition, May 2005. 346
- Eiichi Goto. Monocopy and associative algorithms in an extended LISP. Technical Report

- 74-03, Information Science Laboratories, Faculty of Science, University of Tokyo, 1974. 169
- David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977. doi: 10.1145/359897.359903. 457
- Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI 2002*, pages 282–293. doi: 10.1145/512529.512563. 106
- Chris Grzegorzcyk, Sunil Soman, Chandra Krintz, and Rich Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *5th International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, March 2007, pages 325–340. IEEE Computer Society Press. doi: 10.1109/CGO.2007.20. 209
- Samuel Guyer and Kathryn McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *OOPSLA 2004*, pages 237–250. doi: 10.1145/1028976.1028996. 110, 132, 143
- Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *LFP 1984*, pages 9–17. doi: 10.1145/800055.802017. 277, 294
- Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985. doi: 10.1145/4472.4478. 277, 338, 342, 345
- Lars Thomas Hansen. *Older-first Garbage Collection in Practice*. PhD thesis, Northeastern University, November 2000. 128
- Lars Thomas Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *7th ACM SIGPLAN International Conference on Functional Programming*, Pittsburgh, PA, September 2002, pages 247–258. ACM SIGPLAN Notices 37(9), ACM Press. doi: 10.1145/581478.581502. 128
- David R. Hanson. Storage management for an implementation of SNOBOL4. *Software: Practice and Experience*, 7(2):179–192, 1977. doi: 10.1002/spe.4380070206. 41
- Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA 2003*, pages 388–402. doi: 10.1145/949305.949340. 272
- Timothy Harris. Dynamic adaptive pre-tenuring. In *ISMM 2000*, pages 127–136. doi: 10.1145/362422.362476. 132
- Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *International Conference on Distributed Computing*, Toulouse, France, October 2002, pages 265–279. Volume 2508 of *Lecture Notes in Computer Science*. doi: 10.1007/3-540-36108-1_18. 406
- Pieter H. Hartel. *Performance Analysis of Storage Management in Combinator Graph Reduction*. PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, 1988. 73
- Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Phoenix, AZ, November 1991, pages 33–46. ACM SIGPLAN Notices 26(11), ACM Press. doi: 10.1145/117954.117957. 23, 100, 114
- Barry Hayes. Finalization in the collector interface. In *IWMM 1992*, pages 277–298. doi: 10.1007/BFb0017196. 221
- Barry Hayes. Ephemérons: A new finalization mechanism. In *ACM SIGPLAN Conference*

- on *Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA, October 1997, pages 176–183. ACM SIGPLAN Notices 32(10), ACM Press.
doi: 10.1145/263698.263733. 227
- Laurence Hellyer, Richard Jones, and Antony L. Hosking. The locality of concurrent write barriers. In ISMM 2010, pages 83–92. doi: 10.1145/1806651.1806666. 316
- Fergus Henderson. Accurate garbage collection in an uncooperative environment. In ISMM 2002, pages 150–156. doi: 10.1145/512429.512449. 171
- Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998. xviii, xx, 377, 386, 387, 388, 389, 390, 391, 393, 399
- Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.
doi: 10.1109/71.139204. xvii, 249, 342, 343, 344, 345, 361
- Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufman, April 2008. xxiii, 2, 229, 240, 243, 254, 255, 256
- Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3): 463–492, 1990. doi: 10.1145/78969.78972. 254
- Maurice P. Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture*, San Diego, CA, May 1993, pages 289–300. IEEE Press.
doi: 10.1145/165123.165164. 270, 344
- Maurice P. Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. In *16th International Symposium on Distributed Computing*, Toulouse, France, October 2002, pages 339–353. Volume 2508 of *Lecture Notes in Computer Science*, Springer-Verlag.
doi: 10.1007/3-540-36108-1_23. 374
- Matthew Hertz. *Quantifying and Improving the Performance of Garbage Collection*. PhD thesis, University of Massachusetts, September 2006. 208
- Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In OOPSLA 2005, pages 313–326.
doi: 10.1145/1094811.1094836. 30, 55, 79
- Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In Vivek Sarkar and Mary W. Hall, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005, pages 143–153. ACM SIGPLAN Notices 40(6), ACM Press. doi: 10.1145/1064978.1065028. 9, 108, 110, 156, 208
- Matthew Hertz, Jonathan Bard, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Kirk Kelsey, and Chen Ding. Waste not, want not — resource-based garbage collection in a shared environment. Technical Report TR-951, The University of Rochester, December 2009. doi: 1802/8838. 210
- D. S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, October 1973. doi: 10.1145/362375.362392. 96
- Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA 2003, pages 359–373. doi: 10.1145/949305.949337. 143, 144, 158
- Urs Hölzle. A fast write barrier for generational garbage collectors. In GC 1993. xvi, 197, 198

- Antony L Hosking. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. In ISMM 2006, pages 40–51. doi: 10.1145/1133956.1133963. 30, 340
- Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In GC 1993. 201
- Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993, pages 106–119. ACM SIGOPS Operating Systems Review 27(5), ACM Press. doi: 10.1145/168619.168628. 353
- Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 1992, pages 92–109. ACM SIGPLAN Notices 27(10), ACM Press. doi: 10.1145/141936.141946. 137, 138, 193, 194, 195, 197, 199, 201, 202, 206
- Antony L. Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahmamath. Optimizing the read and write barrier for orthogonal persistence. In Ronald Morrison, Mick J. Jordan, and Malcolm P. Atkinson, editors, *8th International Workshop on Persistent Object Systems (August, 1998)*, Tiburon, CA, 1999, pages 149–159. Advances in Persistent Object Systems, Morgan Kaufmann. 323
- Xianlong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Z. Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In OOPSLA 2004, pages 69–80. doi: 10.1145/1028976.1028983. 52, 170
- Richard L. Hudson. Finalization in a garbage collected world. In GC 1991. 221
- Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In GC 1990. 195
- Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, Palo Alto, CA, June 2001, pages 48–57. ACM Press. doi: 10.1145/376656.376810. 346, 361
- Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5): 223–261, 2003. doi: 10.1002/cpe.712. 346, 351, 361
- Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In IWMM 1992, pages 388–403. doi: 10.1007/BFb0017203. 109, 130, 137, 140, 143, 158, 202, 208
- Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts, September 1991. 118, 138
- R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, November 1982. doi: 10.1002/spe.4380121108. 25, 26
- Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *Transactions on Parallel and Distributed Systems*, 4(9): 1030–1040, 1993. doi: 10.1109/71.243529. xx, xxi, 294, 295, 296, 297, 304, 468
- ISMM 1998, Simon L. Peyton Jones and Richard Jones, editors. *1st International Symposium on Memory Management*, Vancouver, Canada, October 1998. ACM SIGPLAN Notices 34(3), ACM Press. doi: 10.1145/286860.437, 451, 455
- ISMM 2000, Craig Chambers and Antony L. Hosking, editors. *2nd International*

- Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM SIGPLAN Notices 36(1), ACM Press. doi: 10.1145/362422. 434, 440, 442, 453, 456, 457
- ISMM 2002, Hans-J. Boehm and David Detlefs, editors. *3rd International Symposium on Memory Management*, Berlin, Germany, June 2002. ACM SIGPLAN Notices 38(2 supplement), ACM Press. doi: 10.1145/773146. 433, 439, 443, 446, 453
- ISMM 2004, David F. Bacon and Amer Diwan, editors. *4th International Symposium on Memory Management*, Vancouver, Canada, October 2004. ACM Press. doi: 10.1145/1029873. 433, 436, 438, 446, 451, 454, 456, 460
- ISMM 2006, Erez Petrank and J. Eliot B. Moss, editors. *5th International Symposium on Memory Management*, Ottawa, Canada, June 2006. ACM Press. doi: 10.1145/1133956. 438, 439, 444, 449, 455, 456, 461
- ISMM 2007, Greg Morrisett and Mooly Sagiv, editors. *6th International Symposium on Memory Management*, Montréal, Canada, October 2007. ACM Press. doi: 10.1145/1296907. 441, 448, 452, 455
- ISMM 2008, Richard Jones and Steve Blackburn, editors. *7th International Symposium on Memory Management*, Tucson, AZ, June 2008. ACM Press. doi: 10.1145/1375634. 446, 448, 455
- ISMM 2009, Hillel Kolodner and Guy Steele, editors. *8th International Symposium on Memory Management*, Dublin, Ireland, June 2009. ACM Press. doi: 10.1145/1542431. 434, 450, 459
- ISMM 2010, Jan Vitek and Doug Lea, editors. *9th International Symposium on Memory Management*, Toronto, Canada, June 2010. ACM Press. doi: 10.1145/1806651. 443, 455
- ISMM 2011, Hans Boehm and David Bacon, editors. *10th International Symposium on Memory Management*, San Jose, CA, June 2011. ACM Press. doi: 10.1145/1993478. 457
- IWMM 1992, Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, St Malo, France, 17–19 September 1992. Volume 637 of *Lecture Notes in Computer Science*, Springer. doi: 10.1007/BFb0017181. 443, 445, 447, 450
- IWMM 1995, Henry G. Baker, editor. *International Workshop on Memory Management*, Kinross, Scotland, 27–29 September 1995. Volume 986 of *Lecture Notes in Computer Science*, Springer. doi: 10.1007/3-540-60368-9. 448, 460
- Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In ISMM 2002, pages 88–99. doi: 10.1145/512429.512440. 146
- Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997. 152
- Richard Jones and Chris Ryder. A study of Java object demographics. In ISMM 2008, pages 121–130. doi: 10.1145/1375634.1375652. 23, 106, 113, 114
- Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins. xxiv, xxv, 6, 17, 30, 42, 54, 67, 79, 117, 119, 124, 138, 142, 150, 167
- Richard E. Jones and Andy C. King. Collecting the garbage without blocking the traffic. Technical Report 18–04, Computing Laboratory, University of Kent, September 2004. This report summarises King [2004]. 446
- Richard E. Jones and Andy C. King. A fast analysis for thread-local garbage collection

- with dynamic class loading. In *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Budapest, September 2005, pages 129–138. IEEE Computer Society Press. doi: 10.1109/SCAM.2005.1. This is a shorter version of Jones and King [2004]. 107, 109, 145, 146, 159
- H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1): 26–30, July 1979. doi: 10.1016/0020-0190(79)90103-0. 32, 37, 38
- Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In *ISMM 2004*, pages 152–162. doi: 10.1145/1029873.1029892. 132
- JVM 2001. *1st Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001. USENIX. 440, 453
- Tomas Kalibera. Replicating real-time garbage collector for Java. In *7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, Madrid, Spain, September 2009, pages 100–109. ACM Press. doi: 10.1145/1620405.1620420. 405
- Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *30th IEEE Real-Time Systems Symposium*, Washington, DC, December 2009, pages 81–92. IEEE Computer Society Press. doi: 10.1109/RTSS.2009.40. 415
- Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI 2006*, pages 354–363. doi: 10.1145/1133981.1134023. 38, 39, 202, 302, 352, 361
- Taehyoun Kim and Heonshik Shin. Scheduling-aware real-time garbage collection using dual aperiodic servers. In *Real-Time and Embedded Computing Systems and Applications*, 2004, pages 1–17. Volume 2968 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-24686-2_1. 415
- Taehyoun Kim, Naehyuck Chang, Namyun Kim, and Heonshik Shin. Scheduling garbage collector for embedded real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Atlanta, GA, May 1999, pages 55–64. ACM SIGPLAN Notices 34(7), ACM Press. doi: 10.1145/314403.314444. 415
- Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Bounding worst case garbage collection time for embedded real-time systems. In *6th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Washington, DC, May/June 2000, pages 46–55. doi: 10.1109/RTAS.2000.852450. 415
- Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software*, 58(3):247–260, September 2001. doi: 10.1016/S0164-1212(01)00042-5. 415
- Andy C. King. *Removing Garbage Collector Synchronisation*. PhD thesis, Computing Laboratory, The University of Kent at Canterbury, 2004. 145, 446
- Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965. doi: 10.1145/365628.365655. 96
- Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison-Wesley, second edition, 1973. 89, 90, 98
- Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. Technical Report 88.384, IBM Haifa Research Lab., November 1999. 284, 289
- David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *USENIX Summer Conference*, Portland, OR, 1985, pages 489–506. USENIX Association. 100, 152

- H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, 1977, pages 120–131. IEEE Press. doi: 10.1109/SFCS.1977.5. 326, 329
- Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In *IWMM 1992*, pages 404–425. doi: 10.1007/BFb0017204. 52, 53
- Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *International Conference on Parallel Processing (ICPP)*, 1976, pages 50–54. xxi, 326, 327
- Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *Symposium on Interpreters and Interpretive Techniques*, St Paul, MN, June 1987, pages 253–263. ACM SIGPLAN Notices 22(7), ACM Press. doi: 10.1145/29650.29677. 137, 149, 150, 151, 159
- LCTES 2003. *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, June 2003. ACM SIGPLAN Notices 38(7), ACM Press. doi: 10.1145/780732. 431, 453
- Ho-Fung Leung and Hing-Fung Ting. An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):538–543, May 1997. doi: 10.1109/71.598280. 248, 249
- Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA 2001*, pages 367–380. doi: 10.1145/504282.504309. 157, 369
- Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, January 2006. doi: 10.1145/1111596.1111597. 108, 369, 374
- Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS-0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999. 63, 331, 369
- LFP 1984, Guy L. Steele, editor. *ACM Conference on LISP and Functional Programming*, Austin, TX, August 1984. ACM Press. doi: 10.1145/800055. 435, 442, 449, 457
- LFP 1992. *ACM Conference on LISP and Functional Programming*, San Francisco, CA, June 1992. ACM Press. doi: 10.1145/141471. 437, 441
- Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. doi: 10.1145/358141.358147. Also report TM-184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981. 103, 116
- Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. doi: 10.1016/0020-0190(92)90088-D. Also Computing Laboratory Technical Report 75, University of Kent, July 1990. 72
- Boris Magnusson and Roger Henriksson. Garbage collection for control systems. In *IWMM 1995*, pages 323–342. doi: 10.1007/3-540-60368-9_32. 386
- Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *32nd Annual ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005, pages 378–391. ACM SIGPLAN Notices 40(1), ACM Press. doi: 10.1145/1040305.1040336. 346
- Sebastien Marion, Richard Jones, and Chris Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In *ISMM 2007*, pages 67–78. doi: 10.1145/1296907.1296918. 110, 132

- Simon Marlow, Tim Harris, Roshan James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In ISMM 2008, pages 11–20. doi: 10.1145/1375634.1375637. 116, 132, 292, 296, 468
- Johannes J. Martin. An efficient garbage compaction algorithm. *Communications of the ACM*, 25(8):571–581, August 1982. doi: 10.1145/358589.358625. 38
- A. D. Martinez, R. Wachenchauser, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990. doi: 10.1016/0020-0190(90)90226-N. 72
- John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960. doi: 10.1145/367177.367199. xxiii, 2, 18, 29
- John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages I*, pages 173–185. ACM Press, 1978. doi: 10.1145/800025.1198360. xxiii
- Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. IBM Research Report RC24505, IBM Research, 2008. 407, 409
- Phil McGachey and Antony L Hosking. Reducing generational copy reserve overhead with fallback compaction. In ISMM 2006, pages 17–28. doi: 10.1145/1133956.1133960. 126
- Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. Concurrent GC leveraging transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008, pages 217–226. ACM Press. doi: 10.1145/1345206.1345238. 270
- Paul E. McKenney and Jack Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1998. 374
- Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi: 10.1109/TPDS.2004.8. 374
- Maged M. Michael and M.L. Scott. Correction of a memory management method for lock-free data structures. Technical Report UR CSD / TR59, University of Rochester, December 1995. doi: 1802/503. 374
- James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, MIT AI Laboratory, March 1994. doi: 1721.1/6622. 171
- David A. Moon. Garbage collection in a large LISP system. In LFP 1984, pages 235–245. doi: 10.1145/800055.802040. 50, 51, 202, 296
- F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978. doi: 10.1145/359576.359583. 36, 42, 299
- F. Lockwood Morris. On a comparison of garbage collection techniques. *Communications of the ACM*, 22(10):571, October 1979. 37, 42
- F. Lockwood Morris. Another compacting garbage collector. *Information Processing Letters*, 15(4):139–142, October 1982. doi: 10.1016/0020-0190(82)90094-1. 37, 38, 42
- J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle? *IEEE*

- Transactions on Software Engineering*, 18(8):657–673, August 1992.
doi: 10.1109/32.153378. 207
- Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In Mary Lou Soffa, editor, *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008, pages 265–276. ACM SIGPLAN Notices 43(3), ACM Press. doi: 10.1145/1508244.1508275. 10
- John Nagle. Re: Real-time GC (was Re: Widespread C++ competency gap). USENET comp.lang.c++, January 1995. 4
- Scott Nettles and James O'Toole. Real-time replication-based garbage collection. In *PLDI 1993*, pages 217–226. doi: 10.1145/155090.155111. 341, 342, 347, 361, 378
- Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In *IWMM 1992*, pages 357–364.
doi: 10.1007/BFb0017201. 341, 361
- Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, March 2007, pages 68–78. ACM Press.
doi: 10.1145/1229428.1229442. 272
- Gene Novark, Trevor Strohman, and Emery D. Berger. Custom object layout for garbage-collected languages. Technical report, University of Massachusetts, 2006. New England Programming Languages and Systems Symposium, March, 2006. 53
- Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A new approach to parallelising tracing algorithms. In *ISMM 2009*, pages 10–19. doi: 10.1145/1542431.1542434. 262, 263, 264, 265, 298, 304, 305, 468
- Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, FL, October 2009, pages 377–390. ACM SIGPLAN Notices 44(10), ACM Press. doi: 10.1145/1640089.1640117. 293, 468
- OOPSLA 1999. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO, October 1999. ACM SIGPLAN Notices 34(10), ACM Press. doi: 10.1145/320384. 434, 457
- OOPSLA 2001. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tampa, FL, November 2001. ACM SIGPLAN Notices 36(11), ACM Press. doi: 10.1145/504282. 433, 434, 448
- OOPSLA 2002. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, November 2002. ACM SIGPLAN Notices 37(11), ACM Press. doi: 10.1145/582419. 455, 461
- OOPSLA 2003. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, November 2003. ACM SIGPLAN Notices 38(11), ACM Press. doi: 10.1145/949305. 430, 432, 433, 442, 444, 454
- OOPSLA 2004. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004. ACM SIGPLAN Notices 39(10), ACM Press. doi: 10.1145/1028976. 429, 431, 442, 444, 454
- OOPSLA 2005. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, October 2005. ACM SIGPLAN Notices 40(10), ACM Press. doi: 10.1145/1094811. 441, 443, 459

- Yoav Ossia, Ori Ben-Yitzhak, Irit Gof, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *PLDI 2002*, pages 129–140. doi: 10.1145/512529.512546. 284, 285, 288, 296, 304, 474
- Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep GC. In *ISMM 2004*, pages 25–36. doi: 10.1145/1029873.1029877. 321, 352
- Ivor P. Page and Jeff Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986. doi: 10.1109/TC.1986.1676786. 96
- Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynès. Incommunicado: efficient communication for isolates. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, October 1994, pages 262–274. *ACM SIGPLAN Notices* 29(10), ACM Press. doi: 10.1145/582419.582444. 107
- Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988. doi: 10.1145/63039.6304. 194
- Harel Paz and Erez Petrank. Using prefetching to improve reference-counting garbage collectors. In *16th International Conference on Compiler Construction*, Braga, Portugal, March 2007, pages 48–63. Volume 4420 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-71229-9_4. 64
- Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V.T. Rajan. Efficient on-the-fly cycle collection. Technical Report CS-2003-10, Technion University, 2003. 369, 370
- Harel Paz, Erez Petrank, David F. Bacon, Elliot K. Kolodner, and V.T. Rajan. An efficient on-the-fly cycle collection. In *CC 2005*, pages 156–171. doi: 10.1007/978-3-540-31985-6_11. 369
- Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-oriented concurrent garbage collection. In *CC 2005*, pages 121–136. doi: 10.1007/978-3-540-31985-6_9. 369
- Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29(4):1–43, August 2007. doi: 10.1145/1255450.1255453. 67, 369, 372, 373
- E. J. H. Pepels, M. C. J. D. van Eekelen, and M. J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical Report 88-10, Computing Science Department, University of Nijmegen, 1988. 67
- James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977. doi: 10.1145/359605.359626. 96
- Erez Petrank and Elliot K. Kolodner. Parallel copying garbage collection using delayed allocation. *Parallel Processing Letters*, 14(2):271–286, June 2004. doi: 10.1142/S0129626404001878. 284
- Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002, pages 101–112. *ACM SIGPLAN Notices* 37(1), ACM Press. doi: 10.1145/503272.503283. 49
- Pekka P. Pirinen. Barrier techniques for incremental tracing. In *ISMM 1998*, pages 20–25. doi: 10.1145/286860.286863. xvii, 20, 315, 316, 317, 318

- Filip Pizlo and Jan Vitek. Memory management for real-time Java: State of the art. In *11th International Symposium on Object-Oriented Real-Time Distributed Computing*, Orlando, FL, 2008, pages 248–254. IEEE Press. doi: 10.1109/ISORC.2008.40. 377
- Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In *ISMM 2007*, pages 159–172. doi: 10.1145/1296907.1296927. 406, 412
- Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI 2008*, pages 33–44. doi: 10.1145/1379022.1375587. 410, 411, 412
- Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *5th European Conference on Computer Systems (EuroSys)*, Paris, France, April 2010a, pages 69–82. ACM Press. doi: 10.1145/1755913.1755922. 416
- Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010b, pages 146–159. ACM SIGPLAN Notices 45(6), ACM Press. doi: 10.1145/1806596.1806615. 413, 414, 415, 416
- PLDI 1988. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, June 1988. ACM SIGPLAN Notices 23(7), ACM Press. doi: 10.1145/53990. 430, 436
- PLDI 1991. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991. ACM SIGPLAN Notices 26(6), ACM Press. doi: 10.1145/113445. 434, 441, 460
- PLDI 1992. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992. ACM SIGPLAN Notices 27(7), ACM Press. doi: 10.1145/143095. 430, 439
- PLDI 1993. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993. ACM SIGPLAN Notices 28(6), ACM Press. doi: 10.1145/155090. 432, 434, 439, 450
- PLDI 1997. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997. ACM SIGPLAN Notices 32(5), ACM Press. doi: 10.1145/258915. 437, 439
- PLDI 1999. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM SIGPLAN Notices 34(5), ACM Press. doi: 10.1145/301618. 434, 437, 457
- PLDI 2000. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000. ACM SIGPLAN Notices 35(5), ACM Press. doi: 10.1145/349299. 435, 439, 454
- PLDI 2001. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001. ACM SIGPLAN Notices 36(5), ACM Press. doi: 10.1145/378795. 430, 431, 436
- PLDI 2002. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. ACM SIGPLAN Notices 37(5), ACM Press. doi: 10.1145/512529. 433, 442, 450
- PLDI 2006, Michael I. Schwartzbach and Thomas Ball, editors. *ACM SIGPLAN Conference*

- on *Programming Language Design and Implementation*, Ottawa, Canada, June 2006. ACM SIGPLAN Notices 41(6), ACM Press. doi: 10.1145/1133981. 436, 446, 458
- PLDI 2008, Rajiv Gupta and Saman P. Amarasinghe, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, June 2008. ACM SIGPLAN Notices 43(6), ACM Press. doi: 10.1145/1375581. 433, 452
- POPL 1994. *21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 1994. ACM Press. doi: 10.1145/174675. 439, 458
- POPL 2003. *30th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 2003. ACM SIGPLAN Notices 38(1), ACM Press. doi: 10.1145/604131. 431, 434
- POS 1992, Antonio Albano and Ronald Morrison, editors. *5th International Workshop on Persistent Object Systems (September, 1992)*, San Miniato, Italy, 1992. Workshops in Computing, Springer. 455, 460
- Tony Printezis. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In *JVM 2001*. 6, 41, 138
- Tony Printezis. On measuring garbage collection responsiveness. *Science of Computer Programming*, 62(2):164–183, October 2006. doi: 10.1016/j.scico.2006.02.004. 375, 376, 415
- Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *ISMM 2000*, pages 143–154. doi: 10.1145/362422.362480. 23, 151, 326
- Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In *ISMM 2002*, pages 100–105. doi: 10.1145/512429.512436. 22, 73, 143
- Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In *ISMM 2002*, pages 127–138. doi: 10.1145/512429.512446. Sable Technical Report 2002–1 provides a longer version. 147
- Christian Queinnec, Barbara Beaudoin, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, June 1989, pages 224–237. Volume 365 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3540512845_42. 328
- John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993. 105, 121, 195, 201
- Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In *LCTES 2003*, pages 93–102. doi: 10.1145/780732.780745. 415
- J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971. doi: 10.1145/321650.321658. 30
- J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):419–499, July 1974. doi: 10.1145/321832.321846. 30
- J. M. Robson. A bounded storage algorithm for copying cyclic structures. *Communications of the ACM*, 20(6):431–433, June 1977. doi: 10.1145/359605.359628. 90
- J. M. Robson. Storage allocation is NP-hard. *Information Processing Letters*, 11(3):119–125, November 1980. doi: 10.1016/0020-0190(80)90124-6. 93
- Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic distributed garbage collection

- with group merger. In Eric Jul, editor, *12th European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998, pages 249–273. Volume 1445 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/BFb0054095. Also UKC Technical report 17–97, December 1997. 58
- Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL–84–7, Xerox PARC, Palo Alto, CA, July 1985. 4
- Erik Ruf. Effective synchronization removal for Java. In PLDI 2000, pages 208–218. doi: 10.1145/349299.349327. 145
- Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In OOPSLA 2003, pages 326–343. doi: 10.1145/949305.949335. 154, 155, 159
- Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC²: High-performance garbage collection for memory-constrained environments. In OOPSLA 2004, pages 81–98. doi: 10.1145/1028976.1028984. 7, 155, 159
- Konstantinos Sagonas and Jesper Wilhelmsson. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In ISMM 2004, pages 1–12. doi: 10.1145/1029873.1029875. 146
- Konstantinos Sagonas and Jesper Wilhelmsson. Efficient memory management for concurrent programs that use message passing. *Science of Computer Programming*, 62(2): 98–121, October 2006. doi: 10.1016/j.scico.2006.02.006. 146
- Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987. 67
- Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993, pages 106–116. ACM Press. doi: 10.1145/165180.165195. 113
- Robert A. Saunders. The LISP system for the Q–32 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, Cambridge, MA, 1974, pages 220–231. Information International, Inc. 32
- Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3):176–213, 2010. doi: 10.1007/s11241-010-9095-4. 415
- Jacob Seligmann and Steffen Garup. Incremental mature garbage collection using the train algorithm. In Oscar Nierstrasz, editor, *9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995, pages 235–252. Volume 952 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/3-540-49538-X_12. 143
- Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL-TR-88-351. 116, 118, 192, 202
- Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In OOPSLA 2002, pages 13–25. doi: 10.1145/582419.582422. 53
- Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, November 2000, pages 9–17. ACM Press. doi: 10.1145/354880.354883. 385, 412, 415
- Fridtjof Siebert. Limits of parallel marking collection. In ISMM 2008, pages 21–29. doi: 10.1145/1375634.1375638. 276, 277, 303, 474

- Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In ISMM 2010, pages 11–20. doi: 10.1145/1806651.1806654. 280, 282, 283, 304, 385, 412, 474
- Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In ISMM 1998, pages 130–137. doi: 10.1145/286860.286874. 385, 412
- Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *6th International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, 1999, pages 96–102. IEEE Press, IEEE Computer Society Press. doi: 10.1109/RTCSA.1999.811198. 182
- David Siegart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In ISMM 2006, pages 52–63. doi: 10.1145/1133956.1133964. xx, xxi, 52, 295, 296, 297, 304, 468
- Jeremy Singer, Gavin Brown, Mikel Lujan, and Ian Watson. Towards intelligent analysis techniques for object pretenuring. In *ACM International Symposium on Principles and Practice of Programming in Java*, Lisbon, Portugal, September 2007a, pages 203–208. Volume 272 of *ACM International Conference Proceeding Series*. doi: 10.1145/1294325.1294353. 80
- Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. Intelligent selection of application-specific garbage collectors. In ISMM 2007, pages 91–102. doi: 10.1145/1296907.1296920. 6
- Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In POS 1992, pages 11–33. 207
- Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):562–686, July 1985. doi: 10.1145/3828.3835. 92
- Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Bachelor of Science thesis AITR-1417, MIT AI Lab, February 1988. doi: 1721.1/6795. 197
- Sunil Soman and Chandra Krintz. Efficient and general on-stack replacement for aggressive program specialization. In *International Conference on Software Engineering Research and Practice (SERP) & Conference on Programming Languages and Compilers, Volume 2*, Las Vegas, NV, June 2006, pages 925–932. CSREA Press. 190
- Sunil Soman, Chandra Krintz, and David Bacon. Dynamic selection of application-specific garbage collectors. In ISMM 2004, pages 49–60. doi: 10.1145/1029873.1029880. 6, 41, 80
- Sunil Soman, Laurent Daynès, , and Chandra Krintz. Task-aware garbage collection in a multi-tasking virtual machine. In ISMM 2006, pages 64–73. doi: 10.1145/1133956.1133965. 107
- Sunil Soman, Chandra Krintz, and Laurent Daynès. MTM²: Scalable memory management for multi-tasking managed runtime environments. In Jan Vitek, editor, *22nd European Conference on Object-Oriented Programming*, Paphos, Cyprus, July 2008, pages 335–361. Volume 5142 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-70592-5_15. 107
- Daniel Spoonhower, Guy Blelloch, and Robert Harper. Using page residency to balance tradeoffs in tracing garbage collection. In VEE 2005, pages 57–67. doi: 10.1145/1064979.1064989. 149, 150, 152
- James W. Stamos. Static grouping of small objects to enhance performance of a paged

- virtual memory. *ACM Transactions on Computer Systems*, 2(3):155–180, May 1984. doi: 10.1145/190.194. 50
- James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1982. doi: 1721.1/15807. 50
- Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980. 87, 92
- Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975. doi: 10.1145/361002.361005. xvii, 229, 315, 316, 330
- Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976. doi: 10.1145/360238.360247. 315, 316, 318, 323, 326, 335
- Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Stanford University, March 1987. Available as Technical Report CSL-TR-87-324. 27
- Peter Steenkiste. The impact of code density on instruction cache performance. In *16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989, pages 252–259. IEEE Press. doi: 10.1145/74925.74954. 80
- Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM 2000*, pages 18–24. doi: 10.1145/362422.362432. 107, 145, 146, 159
- Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999. 128, 157
- Darko Stefanović and J. Eliot B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *ACM Conference on LISP and Functional Programming*, Orlando, FL, June 1994, pages 43–54. ACM Press. doi: 10.1145/182409.182428. 113
- Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *OOPSLA 1999*, pages 370–381. doi: 10.1145/320384.320425. 128
- Darko Stefanović, Matthew Hertz, Stephen Blackburn, Kathryn McKinley, and J. Eliot Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance*, Berlin, Germany, June 2002, pages 25–36. ACM SIGPLAN Notices 38(2 supplement), ACM Press. doi: 10.1145/773146.773042. 129
- V. Stenning. On-the-fly garbage collection. Unpublished notes, cited by Gries [1977], 1976. 315
- C. J. Stephenson. New methods of dynamic storage allocation (fast fits). In *9th ACM Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983, pages 30–32. ACM SIGOPS Operating Systems Review 17(5), ACM Press. doi: 10.1145/800217.806613. 92
- James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *PLDI 1999*, pages 118–127. doi: 10.1145/301618.301652. 179, 180, 181, 188
- Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In *LFP 1984*, pages 159–166. doi: 10.1145/800055.802032. 73

- Sun Microsystems. Memory management in the Java HotSpot Virtual Machine, April 2006. Technical White Paper. 41, 120
- H. Sundell. Wait-free reference counting and memory management. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005. IEEE Computer Society Press. doi: 10.1109/IPDPS.2005.451.374
- Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), March 2005. 275
- M. Swanson. An improved portable copying garbage collector. ONote 86-03, University of Utah, February 1986. 192
- M. Tadman. Fast-fit: A new hierarchical dynamic storage allocation technique. Master's thesis, University of California, Irvine, 1978. 92
- David Tarditi. Compact garbage collection tables. In *ISMM 2000*, pages 50-58. doi: 10.1145/362422.362437. 179, 180, 181, 182
- Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. In *ISMM 2011*, pages 79-88. doi: 10.1145/1993478.1993491. 355
- S. Thomas, W. Charnell, S. Darnell, B. A. A. Dias, J. G. P. Kramskoy, J. Sextonand, J. Wynn, K. Rautenbach, and W. Plummer. Low-contention grey object sets for concurrent, marking garbage collection. United States Patent Application, 20020042807, 1998. 285, 304
- Stephen P. Thomas. *The Pragmatics of Closure Reduction*. PhD thesis, The Computing Laboratory, University of Kent at Canterbury, October 1993. 170
- Stephen P. Thomas. Having your cake and eating it: Recursive depth-first copying garbage collection with no extra stack. Personal communication, May 1995a. 170
- Stephen P. Thomas. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters*, 56(1):1-7, October 1995b. doi: 10.1016/0020-0190(95)00131-U. 170
- Stephen P. Thomas and Richard E. Jones. Garbage collection for shared environment closure reducers. Technical Report 31-94, University of Kent and University of Nottingham, December 1994. 170, 190
- Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *POPL 1994*, pages 188-201. doi: 10.1145/174675.177855. xxv, 106
- Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3): 245-265, September 2004. doi: 10.1023/B:LISP.0000029446.78563.a4. 148, 159
- David A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9:31-49, January 1979. doi: 10.1002/spe.4380090105. 66
- David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pages 157-167. ACM SIGPLAN Notices 19(5), ACM Press. doi: 10.1145/800020.808261. 61, 63, 103, 106, 116, 119, 120, 130
- David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM distinguished dissertation 1986. MIT Press, 1986. 114

- David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, November 1988, pages 1–17. ACM SIGPLAN Notices 23(11), ACM Press. doi: 10.1145/62083.62085. 114, 116, 121, 123, 137, 138, 140
- David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992. doi: 10.1145/111186.116734. 121, 123, 138
- Maxime van Assche, Joël Goossens, and Raymond R. Devillers. Joint garbage collection and hard real-time scheduling. *Journal of Embedded Computing*, 2(3–4):313–326, 2006. Also published in RTS'05 International Conference on Real-Time Systems, 2005. 415
- Martin Vechev. *Derivation and Evaluation of Concurrent Collectors*. PhD thesis, University of Cambridge, 2007. 331
- Martin Vechev, David F. Bacon, Perry Cheng, and David Grove. Derivation and evaluation of concurrent collectors. In Andrew P. Black, editor, *19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005, pages 577–601. Volume 3586 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/11531142_25. 311, 331
- Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI 2006*, pages 341–353. doi: 10.1145/1133981.1134022. 326, 331
- Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzkky. CGCExplorer: A semi-automated search procedure for provably correct concurrent collectors. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007, pages 456–467. ACM SIGPLAN Notices 42(6), ACM Press. doi: 10.1145/1250734.1250787. 313
- VEE 2005, Michael Hind and Jan Vitek, editors. *1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Chicago, IL, June 2005. ACM Press. doi: 10.1145/1064979. 437, 441, 456
- David Vengero. Modeling, analysis and throughput optimization of a generational garbage collector. In *ISMM 2009*, pages 1–9. doi: 10.1145/1542431.1542433. 123
- Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 29(4): 229–239, April 1980. doi: 10.1145/358841.358852. 92
- Michal Wegiel and Chandra Krintz. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In Susan J. Eggers and James R. Larus, editors, *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008, pages 91–102. ACM SIGPLAN Notices 43(3), ACM Press. doi: 10.1145/1346281.1346294. 107
- J. Weizenbaum. Recovery of reentrant list structures in SLIP. *Communications of the ACM*, 12(7):370–372, July 1969. doi: 10.1145/363156.363159. 60
- Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In Martin Odersky, editor, *18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004, pages 519–542. Volume 3086 of *Lecture Notes in Computer Science*, Springer-Verlag. doi: 10.1007/978-3-540-24851-4_24. 272
- Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for Java. In *OOPSLA 2005*, pages 439–453. doi: 10.1145/1094811.1094845. 272

- Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC Considered Harmful. In *LISP Conference*, Stanford University, CA, August 1980, pages 119–127. ACM Press. doi: 10.1145/800087.802797. 49, 107
- Jon L. White. Three issues in objected-oriented garbage collection. In GC 1990. 139
- Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3), April 2008. doi: 10.1145/1347375.1347389. 415
- Jesper Wilhelmsson. *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*. Licentiate thesis, Uppsala University, May 2005. 146
- Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *ACM SIGPLAN Notices*, 24(5):38–46, May 1989. doi: 10.1145/66068.66070. 116
- Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper. 3, 310, 312, 314
- Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In GC 1993. 139
- Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989a. doi: 10.1145/66068.66077. 197
- Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New Orleans, LA, October 1989b, pages 23–35. ACM SIGPLAN Notices 24(10), ACM Press. doi: 10.1145/74877.74882. 116, 118, 120, 121, 197
- Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective “static-graph” reorganization to improve locality in garbage-collected systems. In PLDI 1991, pages 177–191. doi: 10.1145/113445.113461. 50, 51, 53, 296
- Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In IWMM 1995, pages 1–116. doi: 10.1007/3-540-60368-9_19. 10, 90
- Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Memory allocation policies reconsidered. Unpublished manuscript, 1995b. 96
- David S. Wise. The double buddy-system. Computer Science Technical Report TR79, Indiana University, Bloomington, IN, December 1978. 96
- David S. Wise. Stop-and-copy and one-bit reference counting. Computer Science Technical Report 360, Indiana University, March 1993a. See also Wise [1993b]. 73
- David S. Wise. Stop-and-copy and one-bit reference counting. *Information Processing Letters*, 46(5):243–249, July 1993b. doi: 10.1016/0020-0190(93)90103-G. 460
- David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–359, 1977. doi: 10.1007/BF01932156. 73
- P. Tucker Withington. How real is “real time” garbage collection? In GC 1991. 104, 140
- Mario I. Wolczko and Ifor Williams. Multi-level GC in a high-performance persistent object system. In POS 1992, pages 396–418. 108

- Ming Wu and Xiao-Feng Li. Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. In *IEEE International Parallel and Distribution Processing Symposium (IPDPS)*, Long Beach, CA, March 2007, pages 1–10. doi: 10.1109/IPDPS.2007.370317. 288, 289, 298, 304, 474
- Feng Xian, Witawas Srisa-an, C. Jia, and Hong Jiang. AS-GC: An efficient generational garbage collector for Java application servers. In *ECOOP 2007*, pages 126–150. doi: 10.1007/978-3-540-73589-2_7. 107
- Ting Yang, Emery D. Berger, Matthew Hertz, Scott F. Kaplan, and J. Eliot B. Moss. Autonomic heap sizing: Taking real memory into account. In *ISMM 2004*, pages 61–72. doi: 10.1145/1029873.1029881. 209
- Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990. doi: 10.1016/0164-1212(90)90084-Y. xvii, 316, 317, 318, 323, 326, 378, 466
- Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *OOPSLA 2002*, pages 191–210. doi: 10.1145/582419.582439. 132, 143, 163, 323
- Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. Program-level adaptive memory management. In *ISMM 2006*, pages 174–183. doi: 10.1145/1133956.1133979. 210
- W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987. doi: 10.1109/TSE.1987.233201. 415
- Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990. 124, 125, 323, 393
- Benjamin Zorn. The measured cost of conservative garbage collection. *Software: Practice and Experience*, 23:733–756, 1993. doi: 10.1002/spe.4380230704. 116
- Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California, Berkeley, March 1989. Technical Report UCB/CSD 89/544. 10, 113

索引

注意：如果某一条目在本书中存在特定含义，则对应页码将会加粗，例如 17；条目主要相关内容的所在页码将以斜体表示（如果存在的话），例如 53。另外，索引中标注的页码为英文原书页码，与书中边栏的页码一致。

\$ (记法), 246

A

ABA problem (ABA 问题), 2, 238, 239, 285, 374,
也可参见 Concurrency (并发), hardware
primitive (硬件原语): Compare AndSwap,
LoadLinked 与 StoreConditionally

Abstract concurrent collection (抽象并发回收),
331 ~ 335

addOrigins, 333, 334

collect, 333

expose, 334, 335

mark-sweep (标记-清扫), 332 ~ 335

New, 333, 334

scan, 333

write, 333, 334

write barrier (写屏障), 334

Abstract generational collection (抽象分代回收),
134 ~ 136

Abstract reference counting (抽象引用计数),
82 ~ 85

deferred (延迟), 84

Abstract tracing (抽象追踪), 81 ~ 82

Acquire-release (获取-释放), 参见 Concurrency
(并发), memory consistency (内存一致性)
acquireWork, 278, 281, 283, 287, 288,
290

Adaptive system (自适应系统), 80

generational collection (分代回收), 121 ~ 123,
156

ThruMax algorithm (ThruMax 算法), 123

add, 258 ~ 267, 271

addOrigins, 333

Affinity (亲和性), 参见 Memory affinity (内存亲和性), Processor affinity scheduling (处理器亲和性调度)

Age-based collection (基于年龄的回收), 103,
127, 也可参见 Beltway collector (带式回收器), Generational collection (分代回收),
Older-first collection (中年代优先回收)

Algorithmic complexity (算法复杂度), 78, 也可
参见 specific collection algorithm (特定回收算法)

allocate, 14

Cartesian tree (Cartesian 树), 92

free-list allocation (空闲链表分配)

best-fit (最佳适应), 91

first-fit (首次适应), 89

next-fit (循环首次适应), 91

immix, 153

lazy sweeping (懒惰清扫), 25

real-time collection (实时回收), replicating (副本复制), 380, 382

segregated-fits allocation (分区适应分配), 95

semispace copying (半区复制), 44

sequential allocation (顺序分配), 88, 287

Allocation (分配), 14, 87 ~ 102

alignment constraint (对齐要求), 97 ~ 98

asymptotic complexity (渐进复杂度), 93

bitmap (位图), 92, 100, 102

advantage (优势), 93

boundary tag (边界标签), 98, 168

bump a pointer (阶跃指针), 参见 Sequential
allocation (顺序分配)

cache behaviour (高速缓存相关行为), 97,

- 100, 105
 - concurrent mark-sweep (并发标记-清扫), 324
 - concurrent system (并发系统), 101 ~ 102
 - copying collection (复制式回收), 53
 - different in garbage collected system (垃圾回收系统中的不同之处), 87
 - free-list (空闲链表), 参见 Free-list allocation (空闲链表分配)
 - hash consing (哈希构造), 169
 - heap parsability (堆可解析性), 98 ~ 100, 也可参见 Crossing map (跨越映射)
 - linear (线性), 参见 Sequential allocation (顺序分配)
 - local allocation buffer (本地分配缓冲区), 53, 101 ~ 102, 150, 151, 153, 165, 285, 292, 293, 320, 400
 - performance of various scheme (不同策略的性能), 94
 - run-time interface (运行时接口), 161 ~ 166
 - segregated-fit (分区适应), 参见 Segregated-fits allocation (分区适应分配)
 - sequential(顺序), 参见 Sequential allocation(顺序分配)
 - sequential fit (顺序适应), 参见 Free-list allocation (空闲链表分配)
 - size classe (大小分级), 94
 - size constraint (大小限制), 98
 - space overhead (空间开销), 98
 - stack (栈), 参见 Stack allocation (栈上分配)
 - zeroing (清零), 参见 Zeroing (清零)
 - Allocation colour (分配颜色) (黑色或白色), 参见 Tricolour abstraction (三色抽象), allocation colour (分配颜色)
 - Allocation threshold (分配阈值), 参见 Hybrid mark-sweep, copying (混合标记-清扫、复制)
 - Allocator (分配器), custom (自定义), 3
 - Ambiguous pointer (模糊指针), 参见 Pointer (指针), ambiguous (模糊)
 - Ambiguous root (模糊根), 104, 338, 也可参见 Boehm-Demers-Weiser collector (Boehm-Demers-Weiser 回收器)
 - AMD processor (AMD 处理器), 298
 - Amdahl's law (Amdahl 定律), 303
 - Anthracite (煤灰色), 283
 - Appel, 参见 Concurrent collection (并发回收), read barrier (读屏障)
 - Appel-style collection(Appel 式回收), 参见 Generational collection (分代回收), Appel-style (Appel 式)
 - Arraylet (子数组), 385, 393, 404, 413, 414
 - atomic (原子), 245
 - Atomic operation (原子操作), 15
 - Atomic primitive (原子原语), 参见 Concurrency (并发), hardware primitive (硬件原语)
 - AtomicAdd, 241, 252, 253
 - AtomicDecrement, 241
 - AtomicExchange, 232 ~ 234
 - AtomicIncrement, 241, 365
 - Atomicity (原子性), 参见 Concurrency (并发); Transaction (事务)
 - awk, 58
 - Azul System (Azul 系统), 147, 355, 也可参见 Concurrent copying and compaction (并发复制与整理), Pauseless collector (Pauseless 回收器)
- ## B
- Baker's algorithm (Baker 算法), 277, 337 ~ 338, 377, 384
 - Brooks's variant (Brooks 不变式), 参见 Brooks's indirection barrier (Brooks 间接屏障)
 - Cheney scanning (Cheney 扫描), 338
 - collect, 339
 - copy, 339
 - flip, 339
 - read, 338, 339
 - read barrier (读屏障), 338
 - time and space bound (时空界限), 377
 - Baker's treadmill (Baker 转轮), 参见 Treadmill collector (转轮回收器)
 - barrier, 253, 也可参见 rendezvous barrier (汇聚屏障)
 - Barrier (屏障), 参见 Brooks's indirection barrier (Brooks 间接屏障), Read barrier (读屏障), Stack barrier (栈屏障), Write barrier (写屏障)

Beltway collector (带式回收器), 129 ~ 131, 140,
也可参见 Age-based collection (基于年龄
的回收)

Benchmark (基准程序), 10

Best-fit (最佳适应), 参见 Free-list allocation (空
闲链表分配), best-fit (最佳适应)

BiBoP, 参见 Big bag of pages technique (页簇分
配技术)

Big bag of pages technique (页簇分配技术), 27,
168 ~ 169, 183, 294, 295, 也可参见
Sequential-fits allocation (顺序适应分配),
block-based (基于块)

Bitmapped-fit (位图适应), 参见 Allocation (分
配), bitmap (位图)

Black (黑色), 参见 Tricolour abstraction (三色
抽象)

Black mutator (黑色赋值器), 参见 Tricolour
abstraction (三色抽象), mutator colour (赋
值器颜色)

Black-listing (黑名单), 168

Block-structured heap(块结构堆), 参见 Heap(堆),
block-structured (块结构)

Block (内存块), 12

BMU, 参见 Bounded mutator utilisation (界限赋
值器使用率)

Boehm, 参见 Concurrent collection (并发回收),
insertion barrier (插入屏障)

Boehm-Demers-Weiser collector (Boehm-Demers-
Weiser 回收器), 166, 167, 168, 209, 280

Bookmarking collector (书签回收器), 108, 110,
156 ~ 157, 208

Boot image (引导映像), 132, 171

Bounded mutator utilisation (界限赋值器使用
率), 7

Brooks's indirection barrier (Brooks 间接屏障),
340 ~ 341, 386, 391, 393, 404 ~ 407
Read, 341
Write, 341

Buddy system allocation (伙伴系统分配), 参见
Segregated-fits allocation (分区适应分配),
splitting cell (内存单元分裂)

Buffered reference counting (缓冲引用计数), 参
见 Concurrent reference counting (并发引

用计数), buffered (缓存)

C

C, 104, 106, 161, 162, 165, 168, 170, 171,
182, 185, 188

C++, 3, 59, 104, 170, 185, 188, 340, 346

Boost library (Boost 库), 59, 228

C++0x, 3

destructor (析构函数), 220

Standard Template Library (标准模板库), 3

C4 collector (C4 回收器), 参见 Concurrent copying
and compaction (并发复制与整理), Pauseless
collector (Pauseless 回收器)

C#, 53, 150, 218

Cache behaviour (高速缓存相关行为), 78 ~ 80,
133, 191, 208, 也可参见 Locality (局部性)

alignment effect (对齐效应), 97

allocation (分配), 100, 105

block-based allocation (基于内存块的分配),
95

marking (标记), 21 ~ 22, 27 ~ 29

mutator performance (赋值器性能), 参见 Copying
(复制), improve mutator performance (提
升赋值器性能)

next-fit allocation (循环首次适应分配), 90

partitioning by kind (根据类别进行分区), 105

sequential allocation (顺序分配), 88

threaded compaction (引线整理), 38

Treadmill collector (转轮回收器), 139

Two-Finger algorithm (双指针算法), 34

zeroing (清零), 165, 166

Cache hit (高速缓存命中), 231

Cache line (高速缓存行), 12, 152, 230, 231
dirty (脏), 231

eviction (淘汰), 231, 235

false sharing (伪共享), 232

flushing (刷新), 231, 235

victim (牺牲), 231

Cache miss (高速缓存不命中), 231, 235

Cache (高速缓存)

associativity (相联性)

direct-mapped (直接映射), 231

fully-associative (全相联), 231

- set-associative (组相联), 231
- coherence (一致性), 232 ~ 234
- contention (竞争), 233
- exclusive (排他)/inclusive (包容), 231
- level (分级), 231
- replacement policy (置换策略), 231
- write-back (写回), 231, 232, 235
- write-through (写通), 231
- Canonicalisation table (规范化表), 221, 222
- Card table (卡表), 12, 124, 151, 156, 193, 197 ~ 201, 也可参见 Crossing map (跨越映射)
- concurrent collection (并发回收), 318 ~ 321
- space overhead (空间开销), 198
- summarising card (汇总卡), 201, 319
- two-level (两级), 199
- Write, 198
- Card (卡), 12
- Cartesian tree (Cartesian 树), 参见 Free-list allocation (空闲链表分配), Cartesian tree (Cartesian 树)
- Cell (内存单元), 12
- Cheney scanning (Cheney 扫描), 参见 Baker's algorithm (Baker 算法), Copying (复制); Copying (复制), work list (工作列表)
- Chunked list (内存块链表), 203 ~ 204
- Chunk (内存块), 12, 103
 - for partitioning (分区), 108
- Circular buffer (环状缓冲区), 参见 Concurrent datastructure (并发数据结构), queue (队列), bounded (受限), buffer (缓冲区)
- Closure (闭包), 1, 8, 99, 125, 132, 162, 170, 190, 216, 341
- Cold (冷), 参见 Hot and cold (热与冷)
- collect, 174
 - Baker's algorithm (Baker 算法), 339
 - basic mark-sweep (基本标记-清扫), 18
 - concurrent mark-sweep (并发标记-清扫) (collectEnough), 324
 - concurrent reference counting (并发引用计数)
 - age-oriented (面向年龄), 371
 - buffered (缓冲), 367
 - sliding view (滑动视图), 371
 - copying (复制), 52
 - semispace (半区), 45
 - generational (分代), abstract (抽象), 135
 - incremental tracing (增量追踪) (collectTracing Inc), 333
 - mark-compact (标记-整理), 32
 - real-time collection (实时回收), replicating (副本复制), 382
 - reference counting (引用计数)
 - abstract (抽象), 83
 - abstract deferred (抽象延迟), 84
 - coalesced (合并), 65
 - deferred (延迟), 62
 - tracing (追踪), abstract (抽象), 82
- collectNursery, 135
- Collector (回收器), 12
- Collector thread (回收器线程), 12, 15
- collectorOn/Off, real-time collection (实时回收), replicating (副本复制), 382, 383
- compact
 - mark-compact (标记-整理), 33, 35
- Compressor, 40
- threaded compaction (引线整理), 37
- Compaction (整理)
 - concurrent (并发), 参见 Concurrent copying and compaction (并发复制与整理)
 - incremental (增量), 参见 Hybrid mark-sweep, copying (混合标记-清扫、复制)
 - need for (必要性), 40
 - parallel (并行), 299 ~ 302
- Compressor, 302, 也可参见 Concurrent copying and compaction (并发复制与整理)
- Flood 等, 300 ~ 301, 357
- Compare-and-swap (比较并交换), 237 ~ 238, 243, 也可参见 Concurrency (并发性), hardware primitive (硬件原语), CompareAndSwap
- CompareAndSet, 237
- CompareAndSet2, 242
- CompareAndSetByLLSC, 239
- CompareAndSetWide, 242
- CompareAndSwap, 237, 239, 243, 252, 254, 257, 260, 281, 283, 285, 292, 293, 343 ~ 345, 358, 360, 364 ~ 366, 378, 384, 406, 407, 409 ~ 412
- CompareAndSwap2, 241, 242, 344, 365, 374

- compareAndSwapByLLSC, 239
- CompareAndSwapWide, 241, 242, 257, 406, 412
- compareThenCompareAndSwap, 238
- Comparing collector (回收器之间的比较), 77 ~ 85
- Compiler analysis (编译器分析), 132, 144 ~ 146
 - escape analysis (逃逸分析), 109, 147
- Compiler optimisation (编译器优化), 6, 10, 53, 61, 104, 148, 168, 187, 190, 192, 211, 218, 226, 235, 270, 323, 341, 382, 394
- Compiler replay (编译器重放), 10
- Completeness (完整性), 6, 79, 313
 - Beltway collector (带式回收器), 130, 140
 - partitioned collection (分区回收), 109
- computeLocations, mark-compact (标记-整理), 35
- Concurrency (并发), 56, 也可参见 Lock-free algorithm (无锁算法), Wait-free algorithm (无等待算法)
- consensus algorithm (一致算法), 240, 243, 247, 248
- consensus number (一致数), 240
- happens-before (先于关系), 236
- hardware (硬件), 229 ~ 243
- interconnect (互联), 230 ~ 231
- hardware primitive (硬件原语), 233, 234, 237 ~ 242, 也可参见 primitive (原语)
 - AtomicAdd, 241
 - AtomicDecrement, 241
 - AtomicExchange, 233
 - AtomicIncrement, 241
 - CompareAndSet2, 242
 - CompareAndSetWide, 242
 - CompareAndSet, 237
 - CompareAndSwap, 237
 - CompareAndSwap2, 242
 - CompareAndSwapWide, 242
 - FetchAndAdd, 241
 - LoadLinked, 238
 - multi-word (多字), 240-242
 - relative power (相对强度), 240
 - StoreConditionally, 238
 - TestAndSet, 234
 - time overhead (时间开销), 242 ~ 243
- linearisability (线性化), 参见 Linearisability (线性化)
- livelock (活锁), 244
- memory consistency (内存一致性), 234 ~ 237, 378
 - acquire-release model (获取-释放语义), 236
 - causal (因果), 236
 - memory order (内存顺序), 235
 - program order (程序顺序), 235
 - relaxed (宽松), 237, 265, 331, 373 ~ 374
 - release (释放), 236
 - sequential (顺序), 236
 - strict (严格), 236
- memory fence (内存屏障), 236, 245, 246, 285, 374, 407
- mutual exclusion (互斥), 246 ~ 248, 253, 254
- Peterson's algorithm (Peterson 算法), 247
- progress guarantee (前进保障), 243 ~ 245
 - lock-free (无锁), 243
 - mutual exclusion (互斥), 246
 - obstruction-free (无障碍), 243
 - wait-free (无等待), 243
- shared memory (共享内存), 231
- Concurrent algorithm (并发算法), 也可参见 Concurrent collection (并发回收)
- Concurrent collection (并发回收), 7, 244, 275, 276, 307 ~ 321, 也可参见 specific concurrent collection algorithm (特定并发回收算法)
- abstract (抽象), 参见 Abstract concurrent collection (抽象并发回收)
- collector liveness (回收器存活性), 309, 313, 344
- compaction (整理), 参见 Concurrent copying and compaction (并发复制与整理)
- copying (复制), 参见 Concurrent copying and compaction (并发复制与整理)
- correctness (正确性), 309 ~ 314
- deletion barrier (删除屏障), 317, 318, 328, 329, 345, 391
- Abraham and Patel, 参见 Yuasa
- logging (日志), 331
- Yuasa, 317, 318, 323, 329, 330, 335,

- 378
- handshake (握手), 328, 329, 331, 343, 369 ~ 371, 373, 394
- incremental update approach (增量更新方式), 314
- insertion barrier (插入屏障), 315, 394
 - Boehm 等, 315, 318, 323
 - Dijkstra 等, 315, 318, 323, 326, 329, 330, 335, 340, 370, 386, 413
 - Steele, 315, 318, 323, 326, 335
- lost object problem (对象丢失问题), 310 ~ 312
- mark-sweep (标记-清扫), 参见 Concurrent mark-sweep (并发标记-清扫)
- mostly-concurrent (主体并发), 307, 308
- on-the-fly (即时), 107, 308, 309, 313, 332, 也可参见 specific concurrent collection algorithms (特定并发回收算法)
- progress guarantee (前进保障), 244
- read barrier (读屏障), 314 ~ 321, 323, 346
 - Appel 等, 317, 318, 340
 - Baker, 317, 318, 338, 340
 - Brooks's indirection barrier (Brooks 间接屏障), 参见 Brooks's indirection barrier (Brooks 间接屏障)
 - Compressor, 352 ~ 354
 - Pauseless, 355
 - self-erasing (自删除), 340 ~ 341
 - self-healing (自我修复), 357, 358, 360
- replicating (副本复制), 参见 Concurrent copying and compaction (并发复制与整理), replicating collection (副本复制回收)
- snapshot-at-the-beginning approach (起始快照方式), 314
- termination (结束), 244, 248 ~ 252, 313, 332
- throughput (吞吐量), 313, 345
- work list (工作列表), 319
- write barrier (写屏障), 314 ~ 321, 330, 348
 - mechanism (机制), 318 ~ 320
 - on-the-fly mark-sweep (即时标记-清扫), 329
 - time overhead (时间开销), 342
- Concurrent copying and compaction (并发复制与整理), 337 ~ 361
- C4 collector (C4 回收器), 参见 Concurrent copying and compaction (并发复制与整理), Pauseless collector (Pauseless 回收器)
- compaction (整理), 351 ~ 361, 391, 也可参见 Concurrent copying and compaction (并发复制与整理), Pauseless collector (Pauseless 回收器)
- Compressor, 352 ~ 354, 也可参见 Compaction (整理), parallel (并行), Compressor copying (Compressor 回收器的复制), 337 ~ 351
- mostly-concurrent (主体并发), 参见 Baker's algorithm (Baker 算法)
- fromspace and tospace invariant (来源空间与目标空间不变式), 337, 341, 345, 378
- mostly-copying collection (主体复制式回收), 338 ~ 340
- multi-version copying (多版本复制), 342 ~ 345
- Pauseless collector (Pauseless 回收器), 355 ~ 361
- real-time (实时), 参见 Real-time collection (实时回收), compaction (整理)
- replicating collection (副本复制回收), 289, 341 ~ 351, 也可参见 multi-version copying (多版本复制)
- real-time (实时), 参见 Real-time collection (实时回收), replicating (副本复制)
- Sapphire collector (Sapphire 回收器), 345 ~ 351, 386
 - copyWord, 350
 - Java volatile field (Java volatile 域), 351
 - Write, 参见 Sapphire phases (Sapphire 回收器的各阶段)
 - write barrier (写屏障), 349
- termination (结束), 342, 357, 358
- Concurrent data structure (并发数据结构), 253 ~ 267
 - buffer (缓冲区), 256, 261 ~ 267, 298
 - circular buffer (环状缓冲区), 参见 queue (队列), bounded (受限), buffer (缓冲区)
 - coarse-grained locking (粗粒度锁), 参见 Lock (锁)
 - counting lock (计数锁), 253, 254
 - deque (双端队列), 251, 331
 - fine-grained locking (细粒度锁), 参见 Lock (锁)
 - lazy update (懒惰更新), 255
 - linked-list (链表), 256 ~ 261, 271
 - non-blocking (非阻塞), 255
 - optimistic locking (乐观锁), 参见 Lock (锁)

- queue (队列), 256 ~ 267
 - bounded (受限), 256, 259, 261 ~ 268, 271
 - double-ended (双端), 参见 deque (双端队列)
 - unbounded (无限), 256, 258, 260
- stack (栈), 256 ~ 257
- Concurrent mark-sweep (并发标记-清扫), 323 ~ 335, 391
 - abstract (抽象), 参见 Abstract concurrent collection (抽象并发回收)
 - allocation (分配), 324 ~ 326
 - collect (collectEnough), 324
 - handshake (握手), 328, 329, 331
 - insertion barrier (插入屏障)
 - Dijkstra 等, 331
 - marking (标记), 324, 325, 356
 - New, 324
 - on-the-fly (即时), 328-331
 - marking from stack (从栈开始标记), 328, 358
 - sliding view (滑动视图), 331
 - sweeping (清扫), 330
 - concurrently with marking (与标记过程并发), 326 ~ 328
 - termination (结束), 324 ~ 326, 328, 330
 - triggering collection (触发回收), 323 ~ 324
- Concurrent reference counting (并发引用计数), 363 ~ 374
- age-oriented (面向年龄), 369 ~ 372
 - New, 372
 - Write, 372
- buffered (缓冲), 366 ~ 367
 - collect, 367
 - Write, 367
- coalesced (合并), 368 ~ 369, 也可参见 sliding view (滑动视图)
 - incrementNew, 368, 369
 - Write, 370
- correctness (正确性), 363 ~ 366
- cycle (环), 366 ~ 369, 372 ~ 373, 也可参见 Recycler algorithm (Recycler 算法), asynchronous (异步)
- deferred (延迟), 366, 也可参见 buffered (缓冲)
- generational (分代), 369, 参见 age-oriented (面向年龄)
- handshake (握手), 369 ~ 371, 373
- on-the-fly (即时), 参见 sliding view (滑动视图)
- race (竞争), 363 ~ 366, 370
- Read, 364, 365
- sliding view (滑动视图), 369 ~ 374
 - collect, 371
 - incrementNew, 368, 369, 372
 - New, 372
 - Write, 370, 372, 373
- snapshot of heap (堆快照), 参见 coalesced (合并)
- snooping write barrier (窥探写屏障), 370
- trial deletion (试验删除), 373, 也可参见 Recycler algorithm (Recycler 算法)
- Write, 364, 365
- Connectivity-based collection (基于相关性的回收), 143 ~ 144
 - time overhead (时间开销), 144
- Conservative collection (保守式回收), 30, 104, 105, 也可参见 Boehm-Demers-Weiser collector (Boehm-Demers-Weiser 回收器)
- bitmap marking (位图标记), 23
- copy
 - Baker's algorithm (Baker 算法), 339
 - copying (复制), semispace (半区), 45
- Copy reserve (复制保留区), 参见 Copying (复制), copy reserve (复制保留区)
- Copying (复制), 17, 43 ~ 56, 79, 126, 127, 140, 152, 157, 158, 也可参见 Hybrid mark-sweep, copying (混合标记-清扫、复制)
 - allocate, 44
 - approximately depth-first (准深度优先), 参见 traversal order (遍历顺序)
 - asymptotic complexity (渐进复杂度), 54
 - breadth-first (广度优先), 参见 traversal order (遍历顺序)
 - Cheney scanning (Cheney 扫描), 46 ~ 48, 139, 145, 146, 150, 156, 294, 295, 338, 386
 - parallel (并行), 289
 - termination (结束), 46
- collect, 45, 52
- concurrent (并发), 312, 参见 Concurrent copying

- and compaction (并发复制与整理)
- copy, 45
- copy reserve (复制保留区), 54, 116, 119, 121, 122, 126 ~ 128, 137, 139, 149, 154
- depth-first (深度优先), 参见 traversal order (遍历顺序)
- flip, 45
- flipping (翻转), 139
- forwarding address (转发地址), 参见 Forwarding address (转发地址), copying collection (复制式回收)
- fragmentation solution (去碎片化解决方案), 43
- hierarchical decomposition (层次分解), 参见 Hierarchical decomposition (层次分解)
- improve mutator performance (提升赋值器性能), 49 ~ 53, 106
- Moon's algorithm (Moon 算法), 50, 51
- mostly-copying collection (主体复制式回收), 30, 104
- moving object (移动对象), 55
- optimal object order (最佳对象顺序), 49
- order (顺序), 参见 traversal order (遍历顺序)
- paging behaviour (换页行为), 50
- parallel (并行), 279, 289 ~ 298
 - Blelloch and Cheng, 289 ~ 292
 - card table (卡表), 298
 - channel (通道), 298
 - Cheng, 参见 Blelloch and Cheng
 - Cheng and Blelloch, 参见 Blelloch and Cheng
 - dominant-thread tracing (支配线程追踪), 293 ~ 294
 - Flood 等, 292 ~ 293, 298
 - generational (分代), 298
 - Imai and Tick, 294 ~ 296
 - Marlow 等, 296
 - memory-centric (以内存为中心), 294 ~ 298
 - Oancea 等, 参见 channel (通道)
 - Ogasawara, 参见 dominant-thread tracing (支配线程追踪)
 - processor-centric (以处理器为中心), 289 ~ 294
 - remembered set (记忆集), 298
 - room (工作空间), 382, 参见 Blelloch and Cheng
 - Sieewart and Hirzel, 296 ~ 297
 - termination (结束), 292, 298
- performance (性能), 49, 54
- reordering (重排序), 46 ~ 53, 296
 - online (在线), 52
- replicating collection (副本复制回收), 参见 Concurrent copying and compaction (并发复制与整理), Replicating collection (副本复制回收)
- scan, 45
- semispace (半区), 43 ~ 56, 78, 102, 139, 192
 - allocation issue (分配问题), 53
 - Beltway (带式), 130, 131
 - flipping (翻转), 43
 - overview (溢出), 43
- space overhead (空间开销), 44, 53, 54
- termination (结束), 44
- traversal order (遍历顺序), 46 ~ 53, 139, 也可参见 Hierarchical decomposition (层次分解)
- approximately depth-first (准深度优先), 50, 51
- breadth-first (广度优先), 49, 50
- depth-first (深度优先), 49, 50, 53
- work list (工作列表), 43, 46, 298
 - Cheney scanning (Cheney 扫描), 44 ~ 46
 - implementation (实现), 44 ~ 53
 - stack (栈), 44
- copyWord (Sapphire), 350
- Correctness (正确性), 13, 79, 278
- countingLock, 254
- Critical section (临界区), 参见 Concurrency (并发), mutual exclusion (互斥)
- Crossing map (跨越映射), 101, 182, 199 ~ 201, 也可参见 Card table (卡表); Heap parsing (堆解析)
- search, 200
- Custom allocator (自定义分配器), 参见 Allocator (分配器), custom (自定义)
- Cyclic data structure (环状数据结构), 3, 140, 157, 也可参见 Reference counting (引用计数), cycle segregating (环状分区), 105
- Cyclone, 106

D

- Dangling pointer(悬挂指针), 参见 Pointer(指针),
dangling(悬挂)
- Deadlock(死锁), 2
- Deallocation(释放), explicit(显式), 2 ~ 3
- decide, 243, 247
- decNursery, 136
- decrementOld, reference counting(引用计数),
coalesced(合并), 65
- Demographic(分布特征), of object(对象), 105
- Dependent load(依赖加载), 235
- Deque(双端队列), 参见 Concurrent data structure
(并发数据结构), deque(双端队列),
queue(队列)
- Derived pointer(派生指针), 参见 Pointer(指针),
derived(派生)
- Dijkstra, 参见 Concurrent collection(并发回收),
insertion barrier(插入屏障)
- Dynamic compilation(动态编译), 10, 187
- Dynamic memory allocation(动态内存分配)
description(描述), 1
heap allocation(堆分配), 1

E

- Eden, 参见 Generational collection(分代回收),
generation(分代), young(年轻), nursery(新生区)
- enterRoom, 291
- Ephemeral collection(短暂回收), 参见 Generational
collection(分代回收)
- Ephemeron(寄生对组), 227, 也可参见 Reference(引
用), weak(弱)
- Epoch(时段), 60, 368
ragged(非协同), 366, 402
- Ergonomic collector(Ergonomic回收器), 参见
HotSpot collector(HotSpot回收器)
- Erlang, 146, 也可参见 Functional language(函数
式语言)
- Escape analysis(逃逸分析), 参见 Compiler analysis
(编译器分析), escape analysis(逃逸分析)
- Evacuating(迁移), 43
- Evacuation threshold(迁移阈值), 参见 Hybrid

mark-sweep, copying(混合标记-清扫、
复制)

- ExactVM, 118, 138, 145
- exchangeLock, 233
- exitRoom, 291
- Explicit deallocation(显式释放), 参见 Deallocation
(释放), explicit(显式)
- Explicit freeing(显式释放), 参见 Deallocation(释
放), explicit(显式)
- External pointer(外部指针), 参见 Pointer(指针),
external(外部)

F

- False sharing(伪共享), 参见 Cache line(高速缓
存行), false sharing(伪共享)
- Fast and slow path(快速路径与慢速路径), 80,
164, 165, 191
block-structured allocation(块结构分配), 53
- Fast-fits allocation(快速适应分配), 参见 Free-
list allocation(空闲链表分配), Cartesian
tree(Cartesian树)
- Fat pointer(肥指针), 参见 Pointer(指针), fat(肥)
- FetchAndAdd, 241, 264, 265, 289 ~ 291,
377, 382, 383
- Field(域), 12
- Finalisation(终结), 13, 213 ~ 221, 223, 224,
330
.NET, 220 ~ 221
C++, 220
concurrency(并发), 216, 218 ~ 219
context(上下文), 216
cycle of object(对象环), 218
error(错误), 217
Java, 219 ~ 220
Lisp, 220
order of(顺序), 217 ~ 218, 225 ~ 226
reference counting(引用计数), 214, 216,
220
resurrection(复活), 13, 216
thread(线程), 215 ~ 216
time of(时间), 214 ~ 215
- First-fit(首次适应), 参见 Free-list allocation(空
闲链表分配), first-fit(首次适应)

- Fixed-point (定点解), 81
 - least (最小), 81, 84 ~ 85
 - flip
 - Baker's algorithm (Baker 算法), 339
 - concurrent copying and compaction (并发复制与整理)
 - multi-version copying (多版本复制), 343
 - copying (复制), semispace (半区), 45
 - Floating garbage (浮动垃圾), 7, 79, 106, 113, 313, 327, 328, 330
 - Forwarding address (转发地址), 8, 36, 42, 126, 147, 299, 390, 405, 406
 - allocation (分配), 98
 - Baker's algorithm (Baker 算法), 339
 - compaction (整理), 407 ~ 409
 - Compressor, 38, 301, 302, 353
 - copying collection (复制式回收), 44, 292 ~ 293, 300, 305, 340, 342 ~ 345, 378 ~ 379, 389
 - Lisp 2 (Lisp 2 算法), 34
 - Pauseless, 355, 356, 358, 360
 - Sapphire, 349
 - Two-Finger algorithm (双指针算法), 32
 - Fragmentation (碎片化), 30 ~ 31, 53, 78, 93, 105, 149, 152, 154, 296, 300, 391, 415
 - asymptotic lower bound (渐进下界), 30
 - block-based allocation (基于内存块的分配), 96
 - copying eliminate (通过复制的方式消除), 43
 - external and internal (外部与内部), 95
 - large object (大对象), 137
 - mark-sweep (标记-清扫), 41
 - negative effect (负面作用), 93
 - next-fit allocation (循环首次适应分配), 90
 - old generation (年老代), 126
 - pinned object (被钉住的对象), 186
 - real-time collection (实时回收), 403 ~ 415
 - segregated-fits allocation (分区适应分配), 95
 - Frame (框), 12, 204 ~ 205
 - generational write barrier (分代间写屏障), 205
 - partitioning (分区), 109
 - free, 14
 - Free pointer (空闲指针), 87
 - Free-list allocation (空闲链表分配), 87 ~ 93, 102, 105, 126, 137, 151, 299
 - balanced tree (平衡树), 92
 - best-fit (最佳适应), 91
 - allocate, 91
 - Cartesian tree (Cartesian 树), 92
 - allocate, 92
 - circular first-fit (环状首次适应), 参见 next-fit (循环首次适应)
 - combining multiple scheme (多种策略的结合), 96 ~ 97
 - first-fit (首次适应), 89 ~ 90, 151, 153
 - allocate, 89
 - characteristics of (特征), 90
 - locality (局部性), 54
 - multiple list (多链表), 93
 - next-fit (下次适应), 90 ~ 91, 153
 - allocate, 91
 - cache behaviour (高速缓存相关行为), 90
 - drawbacks of (缺陷), 90
 - space overhead (空间开销), 89
 - speeding up (加速), 92 ~ 93
 - splay tree (splay 树), 92
 - splitting cell (内存单元分裂), 89 ~ 90
 - Fromspace (来源空间), 43
 - Fromspace invariant (来源空间不变式), 参见 Concurrent copying and compaction (并发复制与整理), fromspace and tospace invariant (来源空间与目标空间不变式)
 - Functional language (函数式语言), 73, 115, 161, 190, 也可参见 Erlang, Haskell, ML
 - lazy (懒惰), 66, 125, 132
 - pure (纯), 66
 - Fundamental algorithm for garbage collection (基础垃圾回收算法), 17
- ## G
- Garbage (垃圾), 14
 - Garbage collection (垃圾回收)
 - abstract algorithm (抽象算法), 81 ~ 85, 也可参见 Abstract specific collection algorithm (抽象特定回收算法)
 - advantage (优势), 4

- chaotic nature (复杂性), 11
- comparing algorithm (算法比较), 5 ~ 9
- correctness (正确性), 6
- criticism (批评), 4
- defined (定义), 3-5
- experimental methodology (实验方法), 10
- importance (重要性), 3
- memory leak (内存泄露), 4
- optimisations for specific language (针对特定语言的优化), 8
- partitioning the heap (堆划分), 参见 Partitioned collection (分区回收)
- performance (性能), 9
- portability (可移植性), 9
- safety issue (安全性问题), 6, 309, 313, 344
- scalability (可扩展性), 9
- space overhead (空间开销), 8
- tasks of (任务), 17
- unified theory (统一定律), 80 ~ 85
- Garbage collection of code (针对代码的垃圾回收), 190
- Garbage-First collector (Garbage-First 回收器), 参见 Generational collection (分代回收)
- GC-check point (回收检查点), 188 ~ 189
- GC-point (回收点), 179 ~ 180, 187 ~ 189, 279, 355, 356, 358, 378, 394, 400, 402, 403, 405, 407
- GC-safe point (回收安全点), 参见 GC-point (回收点)
- generateWork, 278, 281, 283, 287, 288, 290
- Generational collection (分代回收), 103, 107, 109, 111 ~ 135, 146, 154, 157, 158, 171, 191, 192, 296, 322, 也可参见 Age-based collection (基于年龄的回收)
- abstract (抽象), 参见 Abstract generational collection (抽象分代回收)
- age recording (年龄纪录), 116 ~ 121
 - aging space (衰老空间), 116 ~ 120
 - high water mark (高水位标记), 120
 - in header word (头部字), 118
- Appel-style (Appel 式), 121 ~ 122, 126, 127, 144, 208, 209
 - Beltway (带式), 130, 131
 - defining (定义), 122
- Beltway (带式), 130, 131
- bucket (桶), 114
- bucket brigade system (桶组系统), 118 ~ 120
- card marking and scanning (卡标记与扫描), 参见 Card table (卡表)
- Eden(新生区), 参见 generation(分代), young(年轻); nursery (新生区)
- full heap collection (整堆回收), 115, 121 ~ 123, 126, 127, 133
- Garbage-First collector (Garbage-First 回收器), 151
- generation (分代), 111
 - old (年老), 126, 145
 - young (年轻), 106, 111, 119, 125, 126, 133, 145
- generational hypothesis (分代假说), 113 ~ 114
 - strong (强), 114
 - weak (弱), 106, 111, 113, 121, 130, 370
- heap layout (堆布局), 114 ~ 117
- inter-generational pointer (分代间指针), 参见 Pointer (指针), inter-generational (分代间)
- large object space (大对象空间), 114
- long-lived data (长寿数据), 41, 132
- major collection (主回收), 参见 full heap collection (整堆回收)
- Mature Object Space (成熟对象空间), 参见 Mature Object Space collector (成熟对象空间回收器)
- measuring time (测量时间), 113
- minor collection (次级回收), 112, 113, 115, 116, 119, 120, 122, 123
- multiple generation (多分代), 115 ~ 116
- nepotism (庇护), 113, 114
- nursery collection (新生区回收), 112, 121, 126, 127, 133, 141, 146, 157
- pretenuring (预分配), 110, 132
- promoting (提升), 110, 111, 112 ~ 117, 121, 127, 130, 132 ~ 134
 - en masse (集体), 116, 122
 - feedback (反馈), 123
- read barrier(读屏障), 参见 Read barrier(读屏障)
- reference counting(引用计数), 参见 Concurrent

- reference counting (并发引用计数), age-oriented (面向年龄)
 - remembered set (记忆集), 参见 Remembered set (记忆集)
 - sequential store buffer (顺序存储缓冲区), 参见 Remembered set (记忆集), Sequential store buffer (顺序存储缓冲区)
 - space overhead (空间开销), 119, 121, 122, 126 ~ 127, 133
 - step (阶), 114, 118, 119, 128, 132, 134, 296
 - survival rate (存活率), 参见 Survival rate (存活率)
 - survivor space (存活对象空间), 119 ~ 121
 - tenuring (提升), 111
 - threatening boundary scheme (危险边界策略), 123
 - throughput (吞吐量), 111
 - ulterior reference counting (超引用计数), 参见 Ulterior reference counting (超引用计数)
 - write barrier (写屏障), 参见 Write barrier (写屏障)
 - Generational hypothesis (分代假说), 参见 Generational collection (分代回收), generational hypothesis (分代假说)
 - Granule (内存颗粒), 11
 - Grey (灰色), 参见 Tricolour abstraction (三色抽象)
 - Grey mutator (灰色赋值器), 参见 Tricolour abstraction (三色抽象), mutator colour (赋值器颜色)
 - Grey packet (灰色工作包), 参见 Marking (标记), parallel (并行), grey packet (灰色工作包)
 - Grey protected (灰色保护), 参见 Object (对象), grey protected (灰色保护)
- H
- Handle (句柄), 1, 104, 184, 185
 - advantage (优势), 1
 - Happens-before (先于关系), 参见 Concurrency (并发), happens-before (先于关系)
 - Hash consing (哈希构造), 参见 Allocation (分配), hash consing (哈希构造)
 - Haskell, 8, 113, 118, 125, 161, 162, 165, 170, 171, 228, 292, 296, 341, 也可参见 Functional language (函数式语言)
 - Heap layout (堆布局), 203 ~ 205, 也可参见 Virtual memory technique (虚拟内存技术) specific collection algorithm (特定回收算法)
 - Heap node (堆节点), 12
 - Heap parsing (堆解析), 20, 166, 168, 170, 182, 也可参见 Allocation (分配), heap parsability (堆可解析性)
 - Heaplet (子堆), 参见 Thread-local heap (线程本地堆)
 - Heap (堆), 11
 - block-structured (块结构), 22, 31, 122, 152, 166 ~ 168, 183, 294 ~ 297
 - relocating (迁移), 205
 - size (大小), 208 ~ 210
 - Hierarchical decomposition (层次分解), 51 ~ 53, 55, 296, 297
 - Hot and cold (热与冷)
 - field (域), 49, 52
 - object (对象), 53
 - HotSpot collector (HotSpot 回收器), 41, 107, 108, 119, 150, 165, 201
 - Ergonomics, 123
 - Hybrid copying, reference-counting (混合复制、引用计数回收), 参见 Ulterior reference counting (超引用计数)
 - Hybrid mark-sweep, copying (混合标记-清扫、复制), 149 ~ 156, 也可参见 Copying (复制), mostly-copying collection (主体复制式回收); Immix; Mark-Copy (标记-复制)
 - Garbage-First collector (Garbage-First 回收器), 150 ~ 151
 - incremental incrementally compacting collector (渐进式增量整理回收器), 150
 - Hybrid reference counting (混合引用计数), mark-sweep (标记-清扫), 366, 370
 - Hyperthreading (超线程), 参见 Multithreading (多线程), simultaneous (同时)
- I
- IBM, 151

Immix, 152 ~ 154, 413
 allocate, 153
 Immortal data (永生数据), 41
 Implementation (实现), difficulty (困难), 79 ~ 80
 incNursery, 136
 Incremental collection (增量回收), 7, 139, 275, 307 ~ 308
 Baker's algorithm (Baker 算法), 参见 Baker's algorithm (Baker 算法)
 incremental incrementally compacting collector (渐进式增量整理回收器), 150
 treadmill collector (转轮回收器), 参见 Treadmill collector (转轮回收器)
 Incremental compaction (增量整理), 参见 Hybrid mark-sweep, copying (混合标记-清扫、复制)
 incrementNew
 concurrent reference counting (并发引用计数)
 coalesced (合并), 369
 sliding view (滑动视图), 369, 372
 reference counting (引用计数), coalesced (合并), 65
 Intel processor (Intel 处理器), 298, 410
 Interesting pointer (回收相关指针), 参见 Pointer (指针), interesting (回收相关)
 Interior pointer (内部指针), 参见 Pointer (指针), interior (内部)

J

J9, 296
 Jamaica (Java 虚拟机), 282, 412, 415
 Java, 106, 113, 124, 125, 145, 151, 152, 162 ~ 165, 169 ~ 172, 179, 185, 190, 201, 209, 215, 216, 218, 219, 223, 224, 234, 282, 296, 330, 341, 346, 356, 360, 391, 405, 406, 412
 pointer equality (指针相等), 347
 Real-Time Specification for (实时规范), 148
 volatile field (volatile 域), 346, 351, 406
 Java Native Interface (Java 原生接口), 104, 394
 Java virtual machine (Java 虚拟机), 116, 138, 也可参见 ExactVM; HotSpot; J9; Jamaica; Jikes RVM; JRockit

 switching collector (切换回收器), 6
 JavaScript, 228
 Jikes RVM (Jiker 虚拟机), 26, 116, 341
 Jonkers's threaded compactor (Jonker 引线整理回收器), 参见 Mark-compact (标记-整理), Jonkers
 JRockit, 138
 JVM, 参见 Java virtual machine (Java 虚拟机)

L

Large address space (大地址空间), 128, 129
 Large object space (大对象空间), 94, 104, 110, 137 ~ 140, 152, 也可参见 Object (对象), large(大); Treadmill collector(转轮回收器)
 generational collection (分代回收), 114, 124
 Large object (大对象), 参见 Object (对象), large(大)
 Lazy sweeping (懒惰清扫), 参见 Sweeping (清扫), lazy (懒惰)
 lazySweep, 25
 Lifetime of object (对象生命周期), 105, 114, 121, 132, 143, 147, 也可参见 Generational collection (分代回收), measuring time (测量时间)
 Linearisability (线性化), 253
 Linearisation point (线性化点), 254 ~ 256
 Lisp, 50, 58, 113, 115, 124, 162, 164, 169, 171, 190, 220, 226, 323, 384, 也可参见 Scheme (策略)
 Lisp 2 algorithm (Lisp 2 算法), 参见 Mark-compact (标记-整理)
 Livelock (活锁), 2, 参见 Concurrency (并发), livelock (活锁)
 Liveness (存活性), 3, 13, 334
 concurrent collector (并发回收器), 参见 Concurrent collection (并发回收), collector liveness (回收器存活性)
 Load balancing (负载均衡), 277 ~ 278, 279, 280, 282, 285, 288, 294, 298 ~ 300, 303 ~ 305
 dynamic (动态), 277
 over-partitioning (细粒度划分), 278
 static (静态), 277

- Load-linked / store-conditionally (链接加载 / 条件存储), 238 ~ 239, 245, 289, 350, 378, 也可参见 Concurrency (并发), hardware primitive (硬件原语): LoadLinked; Store Conditionally
- LoadLinked, 238, 239, 257, 260, 264 ~ 266, 268, 350, 也可参见 Concurrency (并发), hardware primitive (硬件原语), LoadLinked
- Local allocation buffer (本地分配缓冲区), 参见 Allocation (分配), local allocation buffer (本地分配缓冲区)
- Locality (局部性), 78, 279, 296, 297, 304, 也可参见 Cache behaviour (高速缓存相关行为), Paging behaviour (换页行为)
- after copying (复制之后), 46
- copying (复制), 46
- free-list allocation (空闲链表分配), 54
- lazy sweeping (懒惰清扫), 26
- mark-compact (标记 - 整理), 32, 41
- marking (标记), 21 ~ 22
- parallel copying (并行复制), 293
- sequential allocation (顺序分配), 54
- Two-Finger algorithm (双指针算法), 34
- Lock-free algorithm (无锁算法), 244, 249, 251, 255 ~ 257, 260, 261, 263 ~ 268, 271, 280, 342 ~ 345, 374, 406, 410, 也可参见 Concurrency (并发), progress guarantee (前进保障), lock-free (无锁)
- Lock (锁), 232, 254
- coarse-grained locking (粗粒度锁), 254
- counting lock (计数锁), 253, 254
- exchangeLock, 233
- fine-grained locking (细粒度锁), 254, 256, 258, 259, 261 ~ 263
- lock coupling (锁联结), 255
- optimistic locking (乐观锁), 255
- spin (自旋), 232
- AtomicExchange, 233
- TestAndSet, 234
- test then test then set (检测、检测、设置), 240
- test then test-and-set (检测 - 检测并设置), 240
- test-and-set (检测并设置), 232, 234
- test-and-test-and-set (检测 - 检测 - 设置), 233
- testAndSetLock, 234
- testAndTestAndSetExchangeLock, 233
- Logic programming language (逻辑程序语言), 9, 299
- Lost object problem (对象丢失问题), 参见 Concurrent collection (并发回收), lost object problem (对象丢失问题)
- ## M
- Mach operating system (Mach 操作系统), 209
- Machine learning (机器学习), 80
- Major collection (主回收), 参见 Generational collection (分代回收), full heap collection (整堆回收)
- Managed language (托管语言), 1
- Managed run-time system (托管运行时系统), 1
- Managing machine code (托管机器代码), 105
- Many-core processor (众核处理器), 230
- mark
- basic (基本), 24
- basic mark-sweep (基本标记 - 清扫), 19
- marking edge (标记边), 28
- Mark-compact (标记 - 整理), 17, 31 ~ 42, 79, 111, 126, 127, 184, 也可参见 Hybrid (混合); Mark-sweep (标记 - 清扫)
- arbitrary order (任意顺序), 31
- collect, 32
- compact, 33, 35, 37, 40
- compact phase (整理阶段), 31
- compaction (整理)
- cache behaviour (高速缓存相关行为), 38
- one-pass (单次遍历), 38
- three-pass (三次遍历), 34
- two-pass (两次遍历), 32, 37
- Compressor, 38 ~ 41, 127, 302
- computeLocations, 35
- Jonkers, 36 ~ 38, 127
- limitation (限制), 42
- linearising compaction (线性整理), 31
- Lisp 2 algorithm (Lisp 2 算法), 32, 34 ~ 36
- locality (局部性), 32
- mark phase (标记阶段), 31
- parallel compactor (并行整理器), 36

- relocate, 33, 35
- sliding compaction (滑动整理), 31
- threaded compaction (引线整理), 32, 36 ~ 38
 - parallel (并行), 299
- threading pointer (引线指针), 36
- throughput (吞吐量), 34, 41
- Two-Finger algorithm (双指针算法), 32 ~ 34, 36, 184
- updateReferences, 33, 35
- Mark-compact (标记-整理), time overhead (时间开销), 32
- Mark-Copy (标记-复制), 154 ~ 156
 - space overhead (空间开销), 154
- Mark-sweep (标记-清扫), 17 ~ 30, 122, 126, 137, 146, 也可参见 Mark-compact (标记-整理); Marking (标记); Sweeping (清扫)
 - asymptotic complexity (渐进复杂度), 24
 - basic algorithm (基本算法), 18
 - concurrent (并发), 296, 参见 Concurrent mark-sweep (并发标记-清扫)
 - heap layout (堆布局), 20
 - lazy sweeping (懒惰清扫), 参见 Sweeping (清扫), lazy (懒惰)
 - mark phase (标记阶段), 18, 19
 - mutator overhead (赋值器开销), 29
 - space overhead (空间开销), 29, 40, 74
 - sweep, 20
 - sweep phase (清扫阶段), 18, 20
 - termination of basic algorithm (基本算法的结束), 19
 - throughput (吞吐量), 29
 - time overhead (时间开销), 24
 - tricolour marking (三色标记), 20
- Mark/cons ratio (标记/构造率), 6, 54, 111, 129, 144
 - copying and mark-sweep compared (复制式回收与标记-清扫回收之间的比较), 55
- markFromRoots, basic mark-sweep (基本标记-清扫), 19
- Marking (标记), 153
 - asymptotic complexity (渐进复杂度), 276
 - atomicity (原子性), 22
 - bitmap (位图), 22 ~ 24, 151, 299
 - cache behaviour (高速缓存相关行为), 21 ~ 22, 27 ~ 29
- concurrent (并发), 参见 Concurrent mark-sweep (并发标记-清扫)
- incremental (增量), 155
- mark, 28
- mark stack (标记栈), 23, 28
 - overflow (溢出), 24
- marking edge (标记边), 28 ~ 29
- order (顺序, 深度优先或广度优先), 27
- paging (换页), 23
- parallel (并行), 279 ~ 289
 - Barabash 等, 参见 grey packets (灰色工作包)
 - channel (通道), 288 ~ 289
 - Endo 等, 280, 281, 283, 289
 - Flood 等, 278, 280, 282 ~ 284, 288, 289
 - grey packet (灰色工作包), 282, 284 ~ 288, 296
 - Ossia 等, 参见 grey packet (灰色工作包)
 - Siebert [2008], 276
 - Siebert [2010], 280, 282
 - termination (结束), 参见 Concurrent collection (并发回收), termination (结束)
 - Wu and Li, 参见 using channel (使用通道)
- prefetching (预取), 参见 Prefetching (预取), marking (标记)
- time overhead (时间开销), 27
- work list (工作列表), 19, 28, 279, 280
- Marmot, 138
- Mature Object Space collector (成熟对象空间回收器), 109, 130, 140 ~ 143, 151, 194, 208
 - time overhead (时间开销), 143
- Measurement bias (测量偏差), 10
- Memory affinity (内存亲和性), 293
- Memory consistency (内存一致性), 参见 Concurrency (并发), memory consistency (内存一致性)
- Memory fence (内存屏障), 243, 参见 Concurrency (并发), memory fence (内存屏障)
- Memory leak (内存泄露), 2
- Mercury, 171
- Metronome collector (Metronome 回收器), 391 ~ 399, 402, 407, 413, 也可参见 Real-time collection (实时回收), Tax-and-Spend (税

- 收与开支)
 - compaction (整理), 404 ~ 405
 - generational collection (分代回收), 399
 - handshake (握手), 394
 - mutator utilisation (赋值器使用率), 391 ~ 393, 395
 - pause time (停顿时间), 391, 393, 394
 - read barrier (读屏障), 393
 - robustness (鲁棒性), 399
 - root scanning (根扫描), 394
 - scheduling (调度), 394
 - sensitivity (敏感性), 397
 - syncopation, 399
 - time and space analysis (时间与空间分析), 395 ~ 399
 - write barrier (写屏障), 397
 - Minimum mutator utilisation (最小赋值器使用率), 7, 377, 384, 391 ~ 393, 396, 398 ~ 400, 409
 - Minor collection (次级回收), 参见 Generational collection (分代回收), minor collection (次级回收)
 - MIPS processor (MIPS 处理器), 181, 194
 - ML, 8, 106, 113, 121, 124, 125, 146, 148, 162, 165, 170, 171, 201, 208, 209, 228, 329, 330, 342, 384, 也可参见 Functional language (函数式语言)
 - MMTk, 26, 116, 195, 196, 也可参见 JikesRVM
 - MMU, 参见 Minimum mutator utilisation (最小赋值器使用率)
 - Modula-2+, 340, 366
 - Modula-3, 162, 179, 184, 340
 - Moon's algorithm (Moon 算法), 参见 Copying (复制), Moon's algorithm (Moon 算法)
 - Mortality of object (对象死亡率), 105
 - Mostly-concurrent collection (主体并发回收), 参见 Baker's algorithm (Baker 算法)
 - Mostly-copying collection (主体复制式回收), 参见 Copying (复制), mostly-copying collection (主体复制式回收), Concurrent copying and compaction (并发复制与整理), mostly-copying collection (主体复制式回收)
 - Motorola MC68000, 169
 - Moving object (移动对象), 55
 - Multi-tasking virtual machine (多任务虚拟机), 107
 - Multi-version copying (多版本复制), 参见 Concurrent copying and compaction (并发复制与整理), multi-version copying (多版本复制)
 - Multiprocessor (多处理器), 230
 - chip (片上), 230
 - many-core (众核), 230
 - multicore (多核), 230
 - symmetric (对称), 230
 - Multiprogramming (多程序), 230
 - Multiset (多集合), definition and notation (定义与记法), 15
 - Multithreading (多线程), 230
 - simultaneous (同时), 230
 - Mutator (赋值器) 12
 - performance (性能), 参见 Copying (复制), improves mutator performance (提升赋值器性能)
 - thread (线程), 参见 Mutator thread (赋值器线程)
 - Mutator colour (赋值器颜色, 黑色或灰色), 参见 Tricolour abstract (三色抽象), mutator colour (赋值器颜色)
 - Mutator overhead (赋值器开销), 参见 specific collection algorithm (特定回收算法)
 - Mutator suspension (赋值器挂起), 参见 GC-point (回收点)
 - Mutator thread (赋值器线程), 12, 15
 - Mutator utilization (赋值器使用率), 7, 385, 391 ~ 393, 395, 也可参见 Bounded mutator utilization (界限赋值器使用率), Minimum mutator utilisation (最小赋值器使用率)
 - Tax-and-Spend (税收与开支), 400
 - Mutual exclusion (互斥), 参见 Concurrency (并发), mutual exclusion (互斥)
- N
- Nepotism (庇护), 参见 Generational collection (分代回收), nepotism (庇护)
 - .NET, 218, 220

New

- basic mark-sweep (基本标记-清扫), 18
- concurrent mark-sweep (并发标记-清扫), 324
- concurrent reference counting (并发引用计数)
 - age-oriented (面向年龄), 372
 - sliding view (滑动视图), 372
- generational (分代), abstract (抽象), 136
- incremental tracing (增量追踪), 333
- real-time collection (实时回收)
 - replicating (副本复制), 378, 381, 384
 - slack-based (基于间隙), 389
- reference counting (引用计数), abstract (抽象), 83
- tracing (追踪), abstract (抽象), 82
- Next-fit (循环首次适应), 参见 Free-list allocation (空闲链表分配), next-fit (循环首次适应)
- NMT, 参见 Not-Marked-Through (尚未标记过)
- Non-uniform memory access (非一致内存访问), 230, 293 ~ 294, 298, 344, 345
- Not-Marked-Through (尚未标记过), 355 ~ 360
- Notation (记法), 12 ~ 16, 245 ~ 246
- Nursery (新生代), 参见 Generational collection (分代回收), generation (分代), young (年轻)

O

- Object inlining (对象内联), 参见 Scalar replacement (纯值替换)
- Object layout (对象布局), bidirectional (双向), 170
- Object table (对象表), 184 ~ 185
- Object-oriented language (面向对象语言), 169
 - per-class GC method (每类型回收方法), 170, 341
- Object (对象), 12
 - boot image (引导映像), 124, 126
 - cold (冷), 参见 Hot and cold (热与冷), objects (对象)
 - filler (填充), 99 ~ 100
 - grey (灰色), 43
 - grey protected (灰色保护), 312
 - hot (热), 参见 Hot and cold (热与冷), object

(对象)

- immortal (永生), 110, 124, 126, 132 ~ 134
- immutable (不可变), 108, 146
- Java Reference, 参见 Reference (引用), weak (弱)
- large (大), 56, 114, 138, 151, 152, 201, 也可参见 Large object space (大对象空间)
- moving (移动), 139 ~ 140
- pointer-free (无指针), 140
- pointer-free (无指针), 140
- popular (富引用), 143, 156
- type of (类型), 169 ~ 171
- Oblet (子对象), 385, 404, 413, 414
- Obstruction-free algorithm (无障碍算法), 243, 255, 也可参见 Concurrency (并发), progress guarantee (前进保障), obstruction-free (无障碍)
- Old generation (年老代), 参见 Generational collection (分代回收), generation (分代), old (年老)
- Older-first collection (中年代优先回收), 127 ~ 129, 131, 也可参见 Age-based collection (基于年龄的回收)
- Beltway (带式), 130, 131
- deferred older-first (延迟中年优先), 128 ~ 129
- renewal older-first (更新中年优先), 128
- On-stack replacement (栈上替换), 190
- On-the-fly collection (即时回收), 参见 Concurrent collection (并发回收), on-the-fly and specific concurrent collection algorithm (即时回收与特定并发回收算法)
- Online sampling (在线采样), 50, 132
- Opportunistic collector (Opportunistic 回收器), 120, 121
- Over-partitioning (细粒度划分), 参见 Load balancing (负载均衡), over-partitioning (细粒度划分)
- Ownership of object (对象所有权), 3

P

- Page fault (缺页异常), 参见 Paging behaviour (换页行为)
- Page (页), 12

- Paging behaviour (换页行为), 参见 Book marking collector (书签回收器), Copying (复制), paging behaviour (换页行为), Marking (标记), paging (换页)
- Parallel collection (并行回收), 7, 244, 275 ~ 306, 308
- correctness (正确性), 278
- grey packet (灰色工作包), 参见 Marking (标记), parallel (并行), Grey packet (灰色工作包)
- load balancing (负载均衡), 参见 Load balancing (负载均衡)
- memory-centric (以内存为中心), 279
- parallel sweeping(并行清扫), 参见 Sweeping(清扫), parallel (并行)
- parallelisation concern (并行化的注意事项), 276 ~ 277
- processor-centric (以处理器为中心), 279
- synchronisation (同步), 278 ~ 280, 305
- termination (结束), 279, 283 ~ 284, 289, 300
- Partitioned collection (分区回收), 103 ~ 110
- non-age-based scheme (非基于年龄的策略), 137 ~ 159, 也可参见 Connectivity-based collection (基于相关性的回收), Hybrid mark-sweep, copying (混合标记-清扫、复制), Large object space (大对象空间), Mature Object Space collection (成熟对象空间回收), Thread-local collection (线程本地分配)
- partitioning approach (分区策略), 108 ~ 109
- partitioning by age (根据年龄进行分区), 109
- partitioning by availability (根据可用性进行分区), 107 ~ 108
- partitioning by kind (根据类别进行分区), 105
- cache behaviour (高速缓存相关行为), 105
- partitioning by mobility (根据移动性进行分区), 104
- partitioning by mutability (根据易变性进行分区), 108
- partitioning by size (根据大小进行分区), 104
- partitioning by thread (根据线程进行分区), 107
- partitioning for locality (为局部性进行分区), 106 ~ 107
- partitioning for space (为空间进行分区), 104 ~ 105
- partitioning for yield (为效益进行分区), 105 ~ 106
- partitioning to reduce pause time (为降低停顿时间进行分区), 106
- reasons to partition (分区的动因), 103 ~ 108
- reclaiming whole region (回收整个区域), 106, 148
- space overhead (空间开销), 105
- when to partition (何时进行分区), 109 ~ 110
- Pause time (停顿时间), 7, 78, 140, 144, 156, 275, 335, 342, 375, 384, 391, 393, 394, 415
- generational collection (分代回收), 111, 123, 133
- sweeping (清扫), 24
- Pauseless collector (Pauseless 回收器), 参见 Concurrent copying and compaction (并发复制与整理), Pauseless collector (Pauseless 回收器)
- performWork, 278, 281, 283, 287, 288, 290
- perl, 58
- Peterson's algorithm (Peterson 算法), 参见 Concurrency (并发), Peterson's algorithm (Peterson 算法)
- Pinning object (被钉住的对象), 104, 109, 183, 185 ~ 186
- Pointer (指针)
- ambiguous (模糊), 166
- dangling (悬挂), 2, 148
- dangling (悬挂), garbage collection as prevention for (垃圾回收可以避免), 3
- derived (派生), 173, 181, 183 ~ 184, 186
- direction (直接), 125 ~ 126, 128, 144 ~ 146
- external (外部), 185 ~ 186
- fat (肥), 2
- finding (查找), 55, 166 ~ 184
- conservative (保守式), 166 ~ 168
- in code (代码中), 181 ~ 182
- in global root (全局根中), 171
- in object (对象中), 169 ~ 171
- in register (寄存器中), 173 ~ 179
- in stack (栈中), 171 ~ 181

stack map (栈映射), 172 ~ 173, 175, 176, 178, 179

tag (标签), 168-169

hazard (冒险), 374

inter-generational (分代间), 112, 113, 115, 123 ~ 126, 134, 154, 191

interesting (回收相关), 124, 125, 128, 130, 132, 151, 191 ~ 193

interior (内部), 32, 34, 38, 166, 168, 173, 181, 182 ~ 183, 184, 186

shared (共享), 59

smart (智能), 3, 73

reference counting (引用计数), 58, 59

strong (强), 参见 Reference (引用), strong (强)

tidy (整齐), 183

unique (唯一), 58, 73

updates of (更新), 124 ~ 125, 132

weak (弱), 参见 Reference (引用), weak (弱)

Pointers, 13

Poor Richard's Memory Manager (Poor Richard 内存管理器), 210

pop, 257, 268

PowerPC architecture (PowerPC 架构), 262, 298, 407

Precision of concurrent collection (并发回收的精度), 313, 334 ~ 335

Prefetching (预取), 24, 78

allocation (分配), 165, 166

compaction (整理), 36

copying (复制), 51, 53

marking (标记), 21, 27 ~ 29

reference counting (引用计数), coalesced (合并), 64

Two-Finger algorithm (双指针算法), 34

Pretenuring (预分配), 参见 Generational collection (分代回收), pretenuring (预分配)

Primitives (原语), 参见 Concurrency (并发), hardware primitive (并发原语)

Processor affinity scheduling (处理器亲和性调度), 293

Processor (处理器), 229

Profiling (分析), 50, 52

Promptness (及时性), 6, 79, 132, 313

finalisation (终结), 217

generational collection (分代回收), 111

reference counting (引用计数), 73

Pseudo-code for algorithm (算法伪代码), 14

Purple (紫色), 326, 327

push, 257, 268

python, 58, 228

Q

Queue (队列), 参见 Concurrent data structure (并发数据结构), queue (队列)

R

Reachability (可达性), 13, 18

α -reachable (α 可达), 223

finaliser-reachable (终结可达), 213, 223

phantom-reachable (虚可达), 224

softly-reachable (软可达), 223

strongly-reachable (强可达), 221

weakly-reachable (弱可达), 221

Read, 15

Baker's algorithm (Baker 算法), 338, 339

Brooks's indirection barrier (Brooks 间接屏障), 341

concurrent reference counting (并发引用计数), 364, 365

Pauseless collector (Pauseless 回收器), 356

real-time collection (实时回收)

Chicken, 410

Clover, 411

incremental replicating (增量副本复制), 405

replicating (副本复制), 378, 381

slack-based (基于间隙), 389

Staccato, 409

Read barrier (读屏障), 14, 53, 110, 147, 170, 191 ~ 192

Baker's algorithm (Baker 算法), 338

concurrent collection (并发回收), 参见 Concurrent collection (并发回收), read barrier (读屏障)

Metronome, 393

time overhead (时间开销), 323

Real-time collection (实时回收), 245, 375 ~ 415

Blelloch and Cheng, 参见 replicating (副本复制)

Chicken, 410, 412

- Read, 410
- Write, 410
- Clover, 410 ~ 412
 - Read, 411
 - Write, 411
- combined scheduling strategy (多种调度策略的结合), 参见 Tax-and-Spend (税收与开支)
- compaction (整理), 403 ~ 415
- fragmentation (碎片化), 403 ~ 415
- incremental replicating (增量副本复制), 405 ~ 406
 - Read, 405
 - Write, 405
- lock-free (无锁), 参见 Clover, Stopless
- Metronome, 参见 Metronome collector (Metronome 回收器)
- replicating (副本复制), 377 ~ 384, 也可参见 incremental replicating (增量副本复制)
 - allocate, 380, 382
 - collect, 382
 - collectorOn/Off, 382, 383
 - New, 378, 381, 384
 - Read, 378, 381
 - time and space bound (时空界限), 384
 - Write, 378, 381
- scheduling overview (调度策略概述), 376 ~ 377
- Schism, 413 ~ 415
- slack-based (基于间隙), 377, 386 ~ 391, 401
 - collector, 388
 - lazy evacuation (懒惰计算), 387
 - New, 389
 - Read, 389
 - scheduling (调度), 389 ~ 391
 - time overhead (时间开销), 390
 - Write, 389
 - write barrier (写屏障), 386
- Staccato, 407 ~ 410
 - Read, 409
 - Write, 409
- Stopless, 406 ~ 407, 412
- Tax-and-Spend (税收与开支), 399 ~ 403
 - mutator utilisation (赋值器使用率), 400
 - termination (结束), 403
- time-based (基于时间), 377, 也可参见 Metronome collector (Metronome 回收器)
- wait-free (无等待), 参见 Chicken, Staccato
- work-based (基于工作), 377 ~ 385, 391, 也可参见 replicating (副本复制)
- time and space analysis (时空分析), 398 ~ 399
- Real-time system (实时系统), 375 ~ 376
 - hard and soft (硬与软), 375
 - multitasking (多任务), 376
 - schedulability analysis (可调度性分析), 376, 384
 - task (任务), 376
 - WCET, 参见 worst-case execution time (最差执行时间)
 - worst-case execution time (最差执行时间), 376, 384, 385, 390, 413
- Recycler algorithm (Recycler 算法), 68 ~ 70, 157
 - asymptotic complexity (渐进复杂度), 72
 - asynchronous (异步), 72, 366 ~ 373
 - synchronous (同步), 67 ~ 72, 372
- Reference counting (引用计数), 3, 17, 18, 57 ~ 75, 79, 108, 146, 157, 275
 - abstract (抽象), 参见 Abstract reference counting (抽象引用计数)
 - advantage and disadvantage (优势与劣势), 58, 73
 - buffered (缓存), 60, 参见 Buffered reference counting (缓冲引用计数)
 - coalesced (合并), 63 ~ 66, 74, 78, 82, 157
 - collect, 65
 - decrementOld, 65
 - incrementNew, 65
 - Write, 64
 - concurrent (并发), 参见 Concurrent reference counting (并发引用计数)
 - counter overflow (计数溢出), 73
 - cycle (环), 59, 66 ~ 72, 74, 79, 也可参见 Recycler algorithm (Recycler 算法)
 - deferred (延迟), 60, 61 ~ 66, 74, 78, 79, 157, 366
 - collect, 62
 - performance (性能), 63
 - Write, 62

zero count table (零引用表), 61, 366, 372
 finalisation (终结), 214, 216, 220
 generational (分代), 参见 Concurrent reference counting (并发引用计数), age-oriented (面向年龄)
 lazy (懒惰), 60
 limited counter field (受限引用计数域), 72 ~ 74
 partial tracing (局部追踪), 67 ~ 72
 promptness (及时性), 73
 simple (简单), 79
 Write, 58
 sliding view (滑动视图), 60
 smart pointer (智能指针), 参见 Pointer (指针), smart (智能)
 space overhead (空间开销), 72 ~ 74
 sticky count (粘性引用计数), 73
 throughput (吞吐量), 74
 trial deletion (试验删除), 67 ~ 72, 373, 也可参见 Recycler algorithm (Recycler 算法)
 ulterior (超), 参见 Ulterior reference counting (超引用计数)
 weak pointer algorithm (弱指针算法), 67
 weak reference (弱指针), 224
 Reference (引用), 12
 accessing heap allocated object (访问堆中分配的对象), 1
 counting (计数), 参见 Reference counting (引用计数)
 phantom (虚), 224
 soft (软), 223, 360
 strong (强), 221
 weak (弱), 67, 169, 218, 221 ~ 228, 330, 360
 Region inference (区域推断), 参见 Partitioned collection (分区回收), reclaiming whole region (回收整个区域)
 relocate, mark-compact (标记-整理), 33, 35
 Remembered set (记忆集), 113, 124-125, 126, 128, 134, 141, 143, 144, 151, 154, 157, 191, 192, 193, 也可参见 Card table (卡表); Chunked list (内存块链表)
 card table, 参见 Card table (卡表)
 hash table (哈希表), 194 ~ 196
 overflow (溢出), 195 ~ 197

sequential store buffer (顺序存储缓冲区), 参见 Sequential store buffer (顺序存储缓冲区)
 remove, 258 ~ 268, 271
 Remset (记忆集), 参见 Remembered set (记忆集)
 Rendezvous barrier (汇聚屏障), 251 ~ 253, 382
 Replicating collection (副本复制回收), 参见 Copying (复制), concurrent (并发), replicating (副本复制)
 Resurrection of object (对象复活), 参见 Finalisation (终结), resurrection (复活)
 Root (根), 12
 Roots, 13
 rootsNursery, 135
 Run-time interface (运行时接口), 参见 run-time interface (运行时接口)
 managed (托管), 1

S

Sampling (采样), 52, 53
 Sapphire collector (Sapphire 回收器), 参见 Concurrent copying and compaction (并发复制与整理)
 Scalar replacement (纯值替换), 148
 Scalar (纯值), 12
 scan
 copying (复制), semispace (半区), 45
 incremental tracing (增量追踪) (scanTracing Inc), 333
 scanNursery, 135
 Scavenging (筛选), 43, 106
 Scheme, 121, 128, 170, 也可参见 Lisp
 Segregated-fits allocation (分区适应分配), 23, 30, 41, 54, 93 ~ 95, 102, 104, 157, 165, 299
 allocate, 95
 block-based (基于内存块), 95 ~ 96, 102
 cache behaviour (高速缓存相关行为), 95
 space overhead (空间开销), 96
 buddy system (伙伴系统), 96
 splitting cell (内存单元分裂), 96
 Self-adjusting tree (自调整树), 92
 Semispace copying (半区复制), 参见 Copying (复制), semispace (半区)

- Sequence (序列), definition and notation (定义与记法), 15
- Sequential allocation (顺序分配), 31, 44, 54, 87 ~ 88, 102, 105, 126, 133, 151 ~ 153, 157, 164, 165, 294, 301
allocate, 88, 287
cache behaviour (高速缓存相关行为), 88
locality (局部性), 54
- Sequential store buffer (顺序存储缓冲区), 156, 193, 195 ~ 196, 207, 也可参见 Chunked list (内存块链表)
Write, 195
- Sequential-fits allocation (顺序适应分配), 参见 Free-list allocation (空闲链表分配)
- Set definition and notation (集合定义与记法), 15
shared, 245
- Shared pointer (共享指针), 参见 Pointer (指针), shared (共享)
- SITBOL, 41
- Sliding view (滑动视图), 参见 Reference counting (引用计数), coalesced (合并), Concurrent mark-sweep (并发标记-清扫), sliding view (滑动视图)
- Slow path (慢速路径), 参见 Fast and slow path (快速与慢速路径)
- Smalltalk, 50, 58, 113, 115, 168, 171, 185, 193, 228
- Smart pointer (智能指针), 参见 Pointer (指针), smart (智能)
- Space leak (空间泄露), 参见 Memory leak (内存泄露)
- Space overhead (空间开销), 参见 specific collection algorithm (特定回收算法)
- Space usage (空间使用率), 78 ~ 79
- Space (空间), 12, 103
- SPARC architecture (SPARC 架构), 168, 197, 300
- Splay tree (Splay 树), 92
- Staccato collector (Staccato 回收器), 参见 Real-time collection (实时回收), Staccato
- Stack allocation (栈上分配), 147 ~ 148
- Stack barrier (栈屏障), 170, 186 ~ 187, 328, 385
- Stack frame (栈帧), 171 ~ 179, 186 ~ 187
- Stack map (栈映射), 188, 参见 Pointer (指针), finding (查找), stack map (栈映射)
compressing (压缩), 179 ~ 181
- Stacklet (子栈), 187, 384
- Stack (栈), 参见 Concurrent data structure (并发数据结构), stack (栈)
- Standard Template Library (标准模板库), 参见 C++ Standard Template Library (C++ 标准模板库)
- Steele, 参见 Concurrent collection (并发回收), insertion barrier (插入屏障)
- Step (阶), 参见 Generational collection (分代回收), step (阶)
- Stop-the-world collection (万物静止式回收), 275, 276
- Stopless collector (Stopless 回收器), 参见 Real-time collection (实时回收), Stopless
- Store buffer (存储缓冲区), 参见 Write buffer (写缓冲区)
- StoreConditionally, 238, 239, 257, 260, 264 ~ 268, 350, 也可参见 Concurrency (并发), hardware primitive (硬件原语)
- Survival rate (存活率), 106, 116, 118
- Survivor space (存活空间), 参见 Generational Collection (分代回收), survivor space (存活空间)
- sweep, basic mark-sweep (基本标记-清扫), 20
- Sweeping (清扫), 102, 153
allocate, 25
bitmap (位图), 23
concurrent (并发), 参见 Concurrent (并发)
mark-sweep (标记-清扫)
lazy (懒惰), 24 ~ 26, 55, 78, 95, 299, 326
lazySweep, 25
parallel (并行), 299
pause time (停顿时间), 24
sweepNursery, 135

T

- Tax-and-Spend scheduling (税收与开支调度), 参见 Real-time collection (实时回收), Tax-

- and-Spend (税收与开支)
 - Tenuring object (提升对象), 参见 Generation collection (分代回收), promoting objects (提升对象)
 - testAndExchange, 233
 - TestAndSet, 233, 234, 241, 377, 379, 384
 - testAndSetLock, 234
 - testAndTestAndSet, 234
 - testAndTestAndSetExchangeLock, 233
 - testAndTestAndSetLock, 234
 - Thread-local collection (线程本地回收), 144 ~ 147
 - space overhead (空间开销), 147
 - Thread-local heap (线程本地堆), 101, 107, 108, 109, 110, 144 ~ 147, 230, 329
 - Thread-local log buffer (线程本地日志缓冲区), 342
 - Threaded compaction (引线整理), 参见 Mark-compact (标记-整理), threaded compaction (引线整理)
 - Thread (线程), 12, 229
 - Throughput (吞吐量), 6, 77 ~ 78, 208, 也可参见 specific collection algorithm (特定回收算法)
 - ThruMax algorithm (ThruMax 算法), 参见 Adaptive system (自适应系统)
 - Thunk (待计算值), 8, 125, 132, 170
 - Time overhead (时间开销), 参见 specific collection algorithm (特定回收算法)
 - Time measuring (时间测量), 参见 Generational collection (分代回收), measuring time (测量时间)
 - Tospace (目标空间), 43
 - Tospace invariant (目标空间不变式), 参见 Concurrent copying and compaction (并发复制与整理), fromspace and tospace invariant (来源空间与目标空间不变式)
 - Tracing (追踪), 参见 Copying (复制); Mark-compact (标记-整理); Mark-sweep (标记-清扫)
 - abstract (抽象), 参见 Abstract tracing (抽象追踪)
 - Train collector (火车回收器), 参见 Mature Object Space collector (成熟对象空间回收器)
 - Transactional memory (事务内存), 267 ~ 273
 - hardware (硬件), 271
 - Transaction (事务), 267 ~ 269
 - abort and commit (中止与提交), 269
 - transitionRooms, 291
 - Traversal order (遍历顺序), 参见 Copying (复制), traversal order (遍历顺序)
 - Treadmill collector (转轮回收器), 104, 138 ~ 139, 361, 也可参见 Large object space (大对象空间)
 - cache behaviour (高速缓存相关行为), 139
 - space overhead (空间开销), 139
 - time overhead (时间开销), 139
 - Tricolour abstraction (三色抽象), 20 ~ 21, 也可参见 Anthracite (煤灰色), Purple (紫色), Yellow (黄色)
 - abstract concurrent collection (抽象并发回收), 334
 - abstract tracing (抽象追踪), 81
 - allocation colour (分配颜色), 314, 391
 - concurrent collection (并发回收), 309 ~ 313, 352
 - concurrent reference counting (并发引用计数), 370
 - mutator colour (赋值器颜色), 313, 314, 324, 325, 328, 329, 336, 337, 340
 - strong and weak tricolour invariant (强弱三色不变式), 312 ~ 313, 324, 325, 391
 - Treadmill collector (转轮回收器), 138
 - Tuple definition and notation (元组定义与记法), 15
 - Two-finger algorithm (双指针算法), 参见 Mark-compact (标记-整理), two-Finger algorithm (双指针算法)
 - cache behaviour (高速缓存相关行为), 34
 - Type information (类型信息)
 - BiBoP, 参见 Big bag of pages technique (页簇分配技术)
 - Type-accurate collection (类型精确回收), 23, 104
- ## U
- Ulterior reference counting (超引用计数), 157 ~ 158
 - UMass GC Toolkit, 118
 - Unique pointer (唯一指针), 参见 Pointer (指针), unique (唯一)
 - Unreachable (不可达), 14

updateReferences, mark-compact(标记-整理),
33, 35

V

Virtual machine(虚拟机), 参见 Java virtual machine
(Java 虚拟机); Multi-tasking virtual
machine(多任务虚拟机)

Virtual memory(虚拟内存), 也可参见 Paging
behaviour(换页行为)

Virtual memory technique(虚拟内存技术), 140,
165, 193, 202, 203 ~ 210

double mapping(二次映射), 206, 352,
355 ~ 361

guard page(哨兵页), 参见 page protection(页
保护)

page protection(页保护), 140, 195, 205 ~ 208,
317, 340, 352 ~ 361

W

Wait-free algorithm(无等待算法), 243, 255,
261, 271, 301, 302, 410, 415, 也可参
见 Concurrency(并发), progress guarantee
(前进保障), wait-free(无等待)

consensus(一致), 240, 243

Wavefront(回收波面), 20

concurrent collection(并发回收), 310, 334,
338, 340

Weak reference(弱引用), 参见 Reference(引用),
weak(弱)

White(白色), 参见 Tricolour abstraction(三色
抽象)

Wilderness preservation(拓展块保护), 100, 152

Work list(工作列表), 324, 338, 也可参见
Chunked list(内存块链表)

Work pulling(工作拉取), 参见 Work stealing(工
作窃取)

Work pushing(工作推送), 参见 Work sharing(工
作共享)

Work sharing(工作共享), 248 ~ 252, 303 ~ 305,
也可参见 Work stealing(工作窃取)

Work stealing(工作窃取), 248 ~ 252, 267 ~ 268,
279 ~ 284, 292, 299, 303 ~ 305, 342,

也可参见 Work sharing(工作共享)

stealable work queue(可窃取工作队列), 280

termination(结束), 参见 Concurrent collection
(并发回收)

Write, 15, 310, 330

Brooks's indirection barrier(Brooks 间接屏障),
341

card table on SPARC(SPARC 架构之上的卡表),
198

concurrent reference counting(并发引用计数),
364, 365

age-oriented(面向年龄), 372

buffered(缓存), 366, 367

sliding view(滑动视图), 370, 372, 373

concurrent reference counting(并发引用计数),
coalesced(合并), 370

generational(分代), abstract(抽象), 136

generational(分代), with frame(基于帧),
205

incremental tracing(增量追踪), 333

multi-version copying(多版本复制), 344

real-time collection(实时回收)

Chicken, 410

Clover, 411

incremental replicating(增量副本复制), 405

replicating(副本复制), 378, 381

slack-based(基于间隙), 389

reference counting(引用计数)

abstract(抽象), 83

abstract deferred(抽象延迟), 84

coalesced(合并), 64

deferred(延迟), 62

simple(简单), 58

Sapphire Copy phase(Sapphire 回收器的复制阶
段), 349 ~ 351

Sapphire Flip phase(Sapphire 回收器的翻转阶
段), 349, 351

Sapphire Mark phase(Sapphire 回收器的标记
阶段), 348 ~ 349

sequential store buffer(顺序存储缓冲区), 195

Staccato, 409

Write barrier(写屏障), 14, 57, 67, 109, 110,
132, 144, 146 ~ 148, 151, 154, 155,
157, 162, 188, 191 ~ 205

abstract concurrent collection (抽象并发回收),
334

Beltway (带式), 130, 131

card table (卡表), 参见 Card table (卡表)

concurrent collection (并发回收), 参见 Concurrent
collection (并发回收), write barrier (写
屏障)

generational (分代), 112, 115, 119, 124 ~ 126,
133, 134

Metronome, 397

Older-first (中年优先), 128, 129

reference counting (引用计数)

coalesced (合并), 63, 64

deferred (延迟), 62

performance (性能), 61

simple (简单), 58

sequential store buffer (顺序存储缓冲区), 参见
Sequential store buffer (顺序存储缓冲区)

time overhead (时间开销), 202 ~ 203, 323

when required (何时需要), 57

Write buffer (写屏障), 235

Y

Yellow (黄色), 326

Young generation (年轻代), 参见 Generational
collection (分代回收), generation (分代),
young (年轻)

Yuasa, 参见 Concurrent collection (并发回收),
deletion barrier (删除屏障)

Z

zero count table (零引用表), 参见 Reference
counting (引用计数), deferred (延迟),
zero count table (零引用表)

Zeroing (清零), 43, 164, 165 ~ 166

cache behaviour (高速缓存相关行为), 165,
166

垃圾回收算法手册 自动内存管理的艺术

The Garbage Collection Handbook The Art of Automatic Memory Management

在自动内存管理领域，Richard Jones于1996年出版的《Garbage Collection: Algorithms for Automatic Dynamic Memory Management》可谓是一部里程碑式的作品。接近20年过去了，垃圾回收技术得到了非常大的发展，因此有必要将该领域当前最先进的技术呈现给读者。本书汇集了自动内存管理研究和开发者们在过去50年间的丰富经验，在本书中，作者在一个统一的易于接受的框架内比较了当下最重要的回收策略以及最先进的回收技术。

本书从近年来硬件与软件的发展给垃圾回收所带来的新挑战出发，探讨了这些挑战给高性能垃圾回收器的设计者与实现者所带来的影响。在简单的传统回收算法之外，本书还涵盖了并行垃圾回收、增量式垃圾回收、并发垃圾回收以及实时垃圾回收。书中配备了丰富的伪代码与插图，以描述各种算法与概念。

本书特色

- 为1996年《Garbage Collection: Algorithms for Automatic Dynamic Memory Management》一书提供了完整的、最新的、权威的续作。
- 全面讲解并行垃圾回收算法、并发垃圾回收算法以及实时垃圾回收算法。
- 深入剖析某些垃圾回收领域的棘手问题，包括与运行时系统的接口。
- 提供在线数据库支持，包含超过2500条垃圾回收相关文献。



投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

封面设计: 余易 林杉



上架指导: 计算机/程序设计

ISBN 978-7-111-52882-1



9 787111 528821 >

定价: 139.00元